Mobile-Kube: Mobility-aware and Energy-efficient Service Orchestration on Kubernetes Edge Servers

Saeid Ghafouri
Queen Mary University of London
s.ghafouri@qmul.ac.uk

Alireza Karami
Independent Researcher
alireza.karami.m@gmail.com

Danial Bidekani Bakhtiarvan
Independent Researcher
danial.bidekani@gmail.com

Aliakbar Saleh Bigdeli
Independent Researcher
al.ak.saleh@gmail.com

Sukhpal Singh Gill
Queen Mary University of London
s.s.gill@qmul.ac.uk

Joseph Doyle
Queen Mary University of London
j.doyle@qmul.ac.uk

***Abstract*—In recent years Kubernetes has become the de facto standard in the realm of service orchestration. Despite its great benefits, there are still numerous challenges to make it compatible with decentralised cloud computing platforms. One of the challenges of mobile edge computing is that the location of the users is changing over time. This mobility will constantly alter the proximity of the users to their connected services. One solution to this problem is to regularly move services to computing nodes near the users. However, distributing the services in edge nodes only subject to user movements will result in the fragmentation of active nodes. This leads to having active nodes that do not use their full capacity. We have proposed a method called Mobile-Kube to reduce the latency of Kubernetes applications on mobile edge computing devices while maintaining energy consumption at a reasonable level. An experimental framework is designed on top of real-world Kubernetes clusters and real-world traces of mobile users' movements have been used to simulate the users' mobility. Experimental results show that Mobile-Kube can achieve similar energy consumption performance to a heuristic approach that focuses on reducing energy consumption only while reducing the latency of services by 43%.**

***Index Terms*—Resource Management, Energy-Efficiency, Cloud Computing, Edge Computing, Reinforcement Learning**

I. Introduction

Nowadays containerised services are ubiquitous. They provide a modular way of packing a single or set of functionalities in separate isolated environments. The containerisation of services provides agile development of software with DevOps practices by easing testing, modularisation and fully automating the integration and delivery process. From the cloud provider's point of view, containerisation provides lightweight and scalable deployment of cloud services. This interest in using containerised software has led to the implementation of production level container management and orchestration systems. Kubernetes has become the most popular orchestration system as it provides automated solutions to tasks that previously required numerous technologies working together with lots of human provisioning [1].

One of the main challenges with centralised cloud computing clusters is the distance of the infrastructure from the end users [2]. Time-sensitive, real time and location-aware services are some examples of the services that are hard to deploy on distant cloud servers. To address this challenge, the fog and edge computing paradigms [3] have been proposed to distribute the computation so that it is closer to the mobile devices. Mobile edge servers are typically smaller nodes deployed near base stations. Typically users connected to the services on edge nodes can access them directly rather than through the network core. Services deployed on the edge are highly available and usually do not suffer from the communication overhead of centralised clouds [2]. Edge clusters are highly distributed and have limited resources compared to the centralised clouds. Therefore, their resource management is more difficult and requires more automated provisioning methods. Migrating services and Virtual Machines (VMs) subject to users' mobility is one of the objectives that has been the topic of some previous literature [4]–[11]. This problem is studied in a subset of edge computing called **M**obile **E**dge **C**omputing (MEC) that deals with making edge computing paradigms more accessible to mobile device users. The mobile users are typically connected to the core network of fog devices through base stations close to them. The edge server is placed near the base station and can expose the service to users with lower latency. However, mobile users typically are not static users. Thus, the base station to which they have connected changes over time based on their mobility. Previous works in this area have mainly ignored the challenges of deploying their methods on real-world orchestration frameworks like Kubernetes.

The primary focus of architectures such as mobile edge computing [12], cloudlets [13] and fog computing [14] has been minimizing the latency or maximizing the throughput of services to mobile users without considering the energy consumption of the edge cloud and the associated environmental impact. Recently, however, cloud services with large providers such as Microsoft and Amazon have increased their focus on reducing their environmental impact by committing to carbon neutrality by 2030 and 2040 respectively [15], [16]. It is likely that the increasing pressure from government organizations will result in edge clouds also considering their operating energy consumption [17]. There has been considerable research on minimizing the energy consumption of centralized cloud services including work on consolidation [18], geographical load balancing [19], the management of workloads that do not have strict deadlines [20], consensus mechanisms for Vehicular Ad-hoc NETworks (VANETs) [21] and the choice of cloud architecture that should be used to support services [22]. Thus, we propose the Mobile-Kube system that considers both overall energy consumption and the latency of service users to provide a MEC platform with good performance for its services and a reduced environmental impact.

The ***main contributions*** of this research work are:

- We present the Mobile-Kube system that integrates with Kubernetes to reduce the latency of service users on edge clusters based on users' mobility while considering energy consumption. Mobile-Kube models the problem of service placement while maintaining a reasonable energy consumption as an optimization problem and solves it using a reinforcement learning algorithm.
- We compare several reinforcement learning methods to determine the best reinforcement learning algorithm to achieve a reasonable trade-off between the two aforementioned objectives. Our experiments show that IMPALA is the best Reinforcement Learning (RL) method for this scenario based on its data efficiency and fast convergence.
- We evaluate Mobile-Kube on the Google Cloud Platform using a mobile user movement emulator that is based on real-world traces. Our results show that Mobile-Kube can achieve similar energy consumption performance to a heuristic approach that focuses on reducing energy consumption only while reducing the latency of services by 43%.

II. Motivating Scenario

In MEC networks, users are typically connected to a base station that connects them to one of the edge servers. In most scenarios users are typically connected to their closest base station. As the users of mobile edge cloud move around they might transfer from one base station and connect to another base station that is closer as it provides a better signal. The problem is that the service that they are connected to may be closer to the former base station and this can result in increased latency. One solution to this is to move the service from their previous location to a place closer to a new base station that the users of that service are connected to. A greedy algorithm that just moves the service immediately to an empty server closest to the base station can be utilised. However, the problem with that approach is that the new placement might result in switching on a new node which will increase energy consumption. In this work, we try to learn a service placement algorithm that can achieve a balance between keeping the number of active nodes to a minimum while providing a reasonable Quality of Service (QoS) to the mobile users through moving services to nodes closer to the users.

III. Related Works

Mobility Aware Service Orchestration at Edge. With the advent of 5G and Long-Term Evolution (LTE) networks over the past few years, the need for mobility driven service placement methods has become more evident. Ouyang *et al.* [4] have proposed a method for migrating the services to nearby servers to the users while maintaining a reasonable migration cost using Lyapunov optimisation. While it is not evaluated experimentally they still have provided a strong theoretical analysis to achieve an optimised service placement subject to latency and a long term cost budget. Badri *et al.* [23] examined the trade-off between energy consumption and reducing the latency for QoS with service migration through a stochastic optimisation approach. The method is able to handle non-deterministic user movements. The cost of relocating a service is also considered in their simulations. Due to the Markovian nature of the service placement problem reinforcement learning and bandit based methods have also been used in recent years. Wang *et al.* [24] models the problem as a Markov decision process under different scenarios of 1-D and 2-D mobility. Real-world datasets have been used by Wang *et al.* in [7] to show the effectiveness of an offline reinforcement learning based approach to reduce the overall service delay. The most similar papers to our work are [5], [10] which have used different variations of RL for placement of cloud services subject to the user mobility while achieving a trade-off in energy consumption. Tang *et al.* [5] have used a variation of reinforcement learning namely Q-learning to model user movements as a Markov decision process and have considered migration costs into account. Despite their precise modeling of the cost and delay in edge nodes, their implementation is not yet on the real-world production-ready orchestration frameworks like Kubernetes. They have used Checkpoint/Restore In Userspace (CRIU) for implementation of the real-world containers migrations for comparison of migration time between VMs and containers. The resource consumption and delay are modeled and the RL algorithm can maintain a balance between the two objectives of reducing the delay and power consumption [25]. However, their experiments have only been tested in a trace-driven simulation environment and no integration with real-world service orchestration systems is presented. To solve the large state space of the MEC placement problem Brandherm *et al.* and Liu *et al.* [6], [9] have used Multi-Agent version of the RL algorithms in distributed settings. However, they have used different Multi-agent algorithms. The former has used q-learning and the latter uses actor-critic networks.

Kubernetes Limitations Ignored in previous works. Other than the lack of the real-world implementation of the container migration, there is another problem in the theoretical assumptions of the previous works when it comes to deploying them in real-world systems. Kubernetes resource models work through the request and limit model. The request is the amount of the resource reserved for a Kubernetes container in each node and the limit is the maximum resources that could be used from the cluster. The problem with some of the mentioned previous works [8], [23] is that the requested computational resource is reserved from the user side and a new container is started for each user task. However, most of the time in reality a container could be serving many users at the same time. Therefore, the number of containers in the nodes for each service is determined by the cloud provider, not the cloud user. While there have been some efforts to address this problem when autoscaling [26], to our knowledge our approach is the first to address this for MEC applications by considering services with multiple connected users.

Reinforcement Learning for Resource Management. Problems that can be expressed as a sequential decision making process can be modelled as a Markov Decision Processes (MDPs). MDPs are the core mathematical formalization that is used in most of the sequential decision[1] making problems. An MDP is a model of sequential decisions that are an abstraction of an agents' behavior in a fully or partially observed environment. An agent is an entity that makes the actions. For example, in a video game, the player who moves the controller is the agent. At each step of decision-making, the agent receives a reward from the environment after the action has been taken. This reward indicates the value of the taken action. If a learning method is associated with the MDP then this reward is a measure that is used to learn to take better actions over the subsequent steps. The main property of the MDPs is that the action taken at each timestamp depends only on the current state of the environment and not any state(s) before that. Mathematically speaking, an MDP consists of a trajectory of states S, actions A and rewards R in the following order:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \qquad (1)$$

Dynamic resource management can be considered an example of sequential decision making since in almost all of its forms it involves deciding which task should be allocated to which resource. Therefore, it can be expressed as an MDP. This gives us robust modeling of the resource management problem that can be used alongside methods for solving the MDPs to allocate resources effectively. After the success of the DeepRM [27] there is a line of research pursuing the problem of resource allocation with RL to optimize classic system problems. The successes of reinforcement learning in playing games and solving control problems [28] have motivated many researchers to take it to their domain of interest. Some examples are device placement optimization [29], video streaming bitrate adaptation [30], and Internet congestion control [31]. As noted before, the resource allocation problem can be expressed as an MDP. This gives rise to the idea of using reinforcement learning in this context which is naturally a learning model designed to solve MDPs. Other than the modeling capabilities of the RL it is adaptable to the incoming workload. One of the drawbacks of RL methods is that they require a large amount of training data. Thus, the abundance of training historical data in resource allocation logs makes them a good fit for the problem [32].

TABLE I
NOMENCLATURE AND NOTATIONAL CONVENTIONS.

cc

Key Concept	Definition
E	Edge Node Set
m	Number of Nodes
E_i	i^{th} node ($i \in [1, m]$)
E_i^l	Location of node i
E_i^C	CPU capacity of node i
E_i^M	Memory capacity of node i
p_i	Power consumed by node i
p_i^{idle}	Idle power of node i
p_i^{max}	Max power of node i
C	Container Set
n	Number of Containers
C_j	j^{th} container ($j \in [1, n]$)
t	Time interval
T	Total time interval in each training episode
$C_j^l(t)$	Location of container j at time t (($c_j^l(t) \in \{E\}$)
$C_j^r(t)$	CPU requirement of container j at time t
$C_j^m(t)$	Memory requirement of container j at time t
M	Mobile User set
o	Number of mobile users
M_k	k^{th} mobile user ($k \in [1, o]$)
M_k^l	Location of mobile user$k$ at time t
M_k^r	Container request of mobile user k at time t
d_{M_k}	Distance of user M_k from its connected service
d_{total}	Total system delay
d_{net}	Network delay
d_q	Delay due to a lack of available resources
p_{total}^r	Total Power consumption
u_i^t	Resource request of node i at time t
C_t	Total Cost

IV. PROBLEM FORMULATION

Let $E = \{E_1, E_2, \dots, E_m\}$ be the set of edge nodes, $C = \{C_1, C_2, \dots, C_n\}$ be the set of containers which host mobile applications and $M = \{M_1, M_2, \dots, M_o\}$ be the set of mobile users which connect to these applications. Each edge node i has a location E_i^l, a CPU capacity E_i^C and a memory capacity E_i^M. Each container j has a location at time t which is denoted as $C_j^t \in \{E\}$. The resource requirements and allocation can change with time. Thus, let $C_j^r(t)$ and $C_j^m(t)$ represent the containers' CPU resource requirements, the containers' memory resource requirements respectively. We also need to consider the location of the mobile user which can change over time and that they may use different mobile applications at different times. Thus, let $M_k^l(t)$ represent

[1]In the context of reinforcement learning these decisions are referred to as actions.

the location of the mobile user k and $M_k^r(t)$ represent the application that the mobile user k is accessing.

The overall goal of the system is to minimise the energy consumption of edge cloud and maximise the performance of mobile applications. Maximising the performance of the mobile applications is achieved by minimizing the total delay of the system and the migration of containers. Its goal can be formulated as:

$$\text{minimise } C_t = w_1 d_{total} + w_2 p_{total} \tag{2}$$

$$\text{subject to } \sum_{C_j^l \in E_i} C_j^r \leq E_i^C \quad \forall i \tag{3}$$

$$\sum_{C_j^l \in E_i} C_j^m \leq E_i^M \quad \forall i \tag{4}$$

Where w_1 w_2 are weights used to indicate the relative importance of each sub goal. The conditions of the optimisation problem are used to represent the resource capacities of the edge servers.

The total delay is comprised of delays due to the network d_{net} and delays due to a lack of available resources for computation d_q. It can be formulated as:

$$d_{total} = d_{net} + d_q \tag{5}$$

d_{net} will vary depending on the location of the containers and the mobile users.

d_{net} is computed by averaging each individual users' experienced latency:

$$d_{net} = \sum_{k=1}^{o} d_{M_k} / o \tag{6}$$

d_q will always be zero as the optimisation conditions prevent overloading an edge server. Further work will explore relaxing these constraints to achieve better service consolidation and how this affects service performance.

The power p_i consumed by node i is based upon the resource request of the at node $u_i(t)$. The power consumed by node i can be calculated as:

$$p_i(t) = \begin{cases} (p_i^{idle} + (p_i^{max} - p_i^{idle})) \times u_i(t), & \text{if } u_i(t) > 0 \\ 0, & \text{if } u_i(t) = 0 \end{cases} \tag{7}$$

If the utilization is zero then the system switches the node off and it consumes no power which is why $p_i(t) = 0$ if $u_i(t) = 0$. The total power p_{total} can then be calculated as:

$$p_{total} = \sum_{i=1}^{m} p_i(t) \tag{8}$$

V. Proposed Reinforcement Learning Solution

As the user mobility and the movement of services in our problem are sequential decision making problems preserving the Markov [5] property, deep reinforcement learning solutions are a good approach for them. In deep reinforcement learning an agent tries to learn a policy $\pi(s)$ that maximise the discounted reward received from the environment. To achieve this, it first tries to find the value of the states which is the sum of the observed rewards from the starting state until the terminal state $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s]$. Saving all the state values in a table is not feasible in problems with large state-space. In deep reinforcement learning a neural network is used to save all the values of states and at each timestamp of the reinforcement learning algorithm, the weights of the neural network are updated based on the received reward from the environment. In value based reinforcement learning the neural network receives the state of the environment as the input and returns the state-value as the output. A maximisation or a greedy policy step is done afterward to choose the appropriate action from the output state values received from the neural network. However, in policy gradient methods the neural network returns the action directly. [33] The general scheme of the used RL algorithms is shown in Algorithm 1. The $\pi_{\theta_{old/new}}$ on line 6 depends on whether the old or new policy is used based on whether the method is an on policy or off policy RL algorithm. In our case, PG and PPO are on policy and IMPALA is off policy.

Algorithm 1 Migration Algorithm

```
 1: for $iterations = 1, 2, \ldots, N$ do
 2:     for $episodes = 1, 2, \ldots, M$ do
 3:         $R_b \leftarrow 0$
 4:         $R_l \leftarrow 0$
 5:         for $timesteps = 1, 2, \ldots, T$ do
 6:             Run a trajectory $\pi_{\theta_{old/new}}$ for $T$ timesteps
 7:             Compute the timestep latency reward $r_l$
 8:             Compute the timestep binpacking reward $r_b$
 9:         end for
10:         $R_b \leftarrow R_b + r_b$
11:         $R_l \leftarrow R_l + r_l$
12:     end for
13:     $R \leftarrow w_1 R_b + w_2 R_l$
14:     Optimize the RL Agent policy $\pi$ based on R
15:     Move the containers based on $\pi$
16: end for
```

Agent We have used three different RL agents in our experiments:

- **PG** Vanilla Policy Gradient (PG) is considered as the basis of all policy gradient RL algorithms. At each iteration of the standard policy gradient method, an episode τ (or a batch of episodes) is performed. Each timestep $t \in T$ of an episode is comprised of a state s_t, action a_t and a reward received from the environment for that state-action pair $r\left(\mathbf{s}_t^i, \mathbf{a}_t^i\right)$. The series of all these state, action and reward triplets constitute a full episode trajectory $\tau \sim \{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\}$. The sum of all the rewards in a sample episode $r(\tau) = \sum_t r\left(\mathbf{s}_t^i, \mathbf{a}_t^i\right)$ is used to update the policy network π sets of parameters θ. To reach this goal first an objective function is computed using the logarithm of the gradients of the policy neural network $\log \pi_\theta(\tau)$ based on the policy gradient theorem

[33].

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau) r(\tau) \tag{9}$$

A gradient ascent step updates the weights of the policy network:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{10}$$

- **PPO** RL algorithms are of high variance by nature. This means that the output actions can change in different training iterations depending on the received random sample. The Proximal Policy Optimization (PPO) algorithm solves this issue by constraining the next action within a certain range at each timestep. It does so by clipping the objective function and using a surrogate objective function instead of the equation 10 [34].
- **IMPALA** is a distributed deep RL method [35]. It proposes a distributed training method for one of the former Deep RL methods named A2C. The naive policy gradient method is a high variance method and it is not stable for environments with high fluctuation in the reward signal. This is because the sum of rewards $r(\tau)$ used in equation 9 are only sampled from a single episode. But due to environment dynamics, this single episode return might not be a good indicator of the states' worthiness. One of the techniques used for reducing the variance is to have another neural network called critic which acts as an estimator for the reward function. In each timestep, the parameters ϕ of another neural network are also trained for estimating a value function:

$$V^{\pi}\left(\mathbf{s}_t\right) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r\left(\mathbf{s}_{t'}, \mathbf{a}_{t'}\right) \mid \mathbf{s}_t\right] \tag{11}$$

This value function will be a better estimate since it is an approximation based on a series of episodes rather than a single episode substituting the $r(\tau)$ with $V^{\pi}\left(\mathbf{s}_i, \mathbf{a}_i\right)$ the equation 9 will update to:

$$\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta\left(\mathbf{a}_i \mid \mathbf{s}_i\right) V^{\pi}\left(\mathbf{s}_i, \mathbf{a}_i\right) \tag{12}$$

The update rule of the algorithm will stay the same as equation 10. At each timestep of the A2C, it distributes the computation of the advantage values into several learners. IMPALA improves the training procedure by distributing the learner into several learners. In former distributed versions of the A2C named A3C, the updating happens through passing the gradient to a central learner, however, this can produce a large communication overhead. Instead of the gradients, IMPALA directly sends the trajectory of experiences received from multiple actors to the central learner. The distributed actors continuously update their policy with the latest updated policy in the learner and then send them to the learner. This approach increases the exploration and throughput rate. Each of the learners does the actions based on its version of the policy network not the latest updated version of the policy. To solve this problem, IMPALA uses a correction step using importance sampling called V-trace. Training another policy μ other than the policy π that the agent is using to act in the environment is referred to as off policy methods. To fix the inconsistency of the actors' policy μ with the learner policy π, in IMPALA the value function in Eq 11 is substituted with another value function with importance sampling to make the learners' policy consistent with the actors' policy.

States. Each state is derived by concatenation of two arrays 1. An array of size n, $U =< u_1, u_2, ..., u_n >$ which the indexes are the users' id and each entry value indicates the corresponding user connected station id and is a value from the set of all available stations $u_i \in [s_1, s_2, ..., s_m]$. 2. Another array $C =< c_1, c_2, ..., c_m >$ which the indexes represent the containers and the items represent the node that the container is placed on, and the values are from $c_i \in [n_1, n_2, ..., n_k]$. All the discrete values are then encoded using a one-hot encoding before being fed to the RL model. The final observation will be the concatenation of these two parts $O = U \bigcup C$.

Actions. The action map is represented as an array of the size of the containers where each of the indices represents one of the hosts and each of the values at that index represents the id of the host that this container is placed at the next timestamp. This is exactly like the U part of the observations.

Reward. The reward at each timestep is calculated per each objective according to Equation 4. For minimizing the network latency we have $R_l = 1/d_{total}$, and for maximizing the number of empty servers we set the binpacking objective to $R_b = p_{total}$. Both of the values are then normalised according to the network size (see Section VII for more information). The two values are then summed up according to two weights w_1 and w_2.

$$R = w_1 R_b + w_2 R_l \tag{13}$$

RL helper. Due to the large state space of the problem, the RL agent will face many illegal actions that try to place containers to hosts without enough space. If we want to end the training episode every time we face an illegal action then we stop the agent from learning longer episodes and it makes the training slow. To solve this problem during the training, we assigned a negative reward to the illegal actions but we continued the simulator traces to the next timestep. In the test phase on the real-world Kubernetes servers, if an illegal action is received, we skip that action and keep the servers at their place until the next timesteps' action.

Policy Networks. We have used a similar structure for the policy neural network across all the reinforcement learning agents. We have used a two-layer fully-connected network with 64 neurons at each layer. We chose this simple architecture as we did not see a meaningful difference in using more complicated architectures when we experimentally evaluated them.

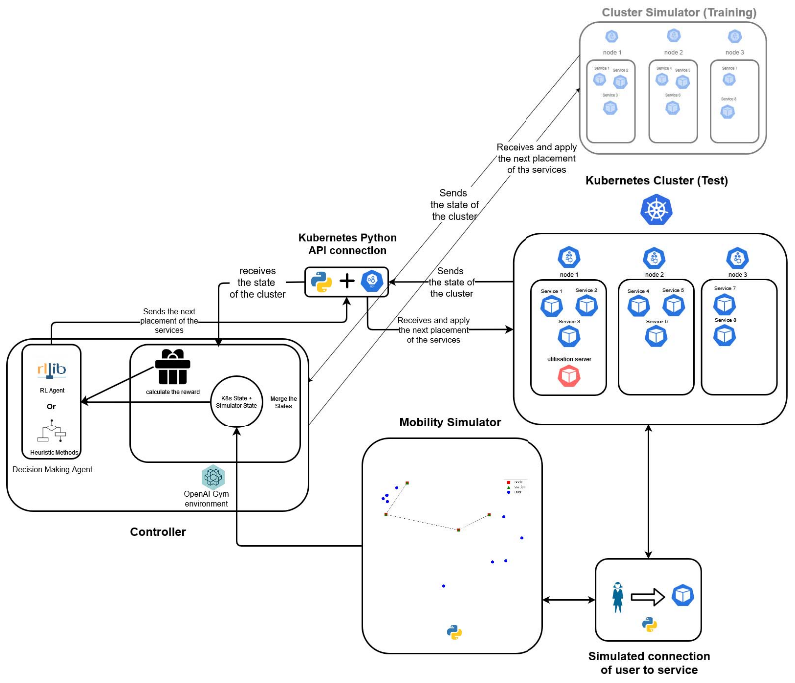
Cluster Simulator (Training)
node 1
node 2
node 3
Receives and apply the next placement of the services
Sends the state of the cluster
Kubernetes Cluster (Test)
Kubernetes Python API connection
receives the state of the cluster
Sends the state of the cluster
Receives and apply the next placement of the services
Sends the next placement of the services
Service 1
Service 2
Service 3
Service 4
Service 5
Service 6
Service 7
Service 8
utilisation server
calculate the reward
K8s State + Simulator State
Merge the States
RL Agent
Or
Heuristic Methods
Decision Making Agent
OpenAI Gym environment
Controller
Mobility Simulator
Simulated connection of user to service

Fig. 1. The design of the system including of five parts (a) the controller in charge of making decisions about the placement of the services (b) The real world Kubernetes cluster (c) Kubernetes Python API connection to the Kubernetes cluster (d) Python mobility simulator (e) Simulated connection of the users to the services.

VI. System Design

We have designed a complete end to end experimental setup. The implementation of the Kubernetes side is all done in real-world Google Cloud cluster. However, the user mobility side is simulation based on real-world data. This is due to the difficulty of having real-world experiments with real moving mobile users.

Kubernetes Internal Structure. Kubernetes clusters are deployed with a number of nodes. *Nodes* can be considered as the equivalent of servers in other forms of computing clusters. A node could be a bare metal server or virtual machine of any kind. Once a computing resource is defined as a Kubernetes node, all the nodes of different types will look the same from the users' perspective. *Containers* in Kubernetes are the minimum isolation level of the applications, but the smallest Kubernetes abstraction for representing containers is another entity called a *pod*. Pods are the smallest deployable unit that presents a set of containers that share the same networking interface. All the containers inside a pod are always co-located in the same cluster node. Pods are ephemeral objects and are replaced and rescheduled constantly during their lifetime. Kubernetes *services* are a way of building a consistent representation of the Kubernetes pods networking for accessing them. This is different from the concept of the service we have used in this work, it is just the Kubernetes internal networking tool that we have used to expose our applications which call a unit of them a service. The state of the cluster and all the internal Kubernetes communications are done through the Kubernetes API server. The API server exposes the Kubernetes API using a rest API. In order to be able to interact with the API server, there are many options like the Kubernetes CLI named kubectl or other client APIs available in multiple languages. We have used the python client API for interacting with the API server.

Our definition of Services. Usually, real world cloud services are made from a set of containers and stateful and stateless microservices. For example, a streaming service consists of a database system, authentication system, video analytic service, and many other small decoupled modular objects. For experimental purposes, we limit the definition of a service to a single pod containing one single container inside it with a Flask Python app for generating load on the container CPU and RAM. The Flask Python app is exposed to the outside world using Kubernetes' service.

Load Generation Module. To emulate artificial load on the services, we have used a model with two containerised Flask applications. One of them called the utilisation server is deployed as a single Flask application in one arbitrary server outside the cluster and the other one is an application running on all the services. At each timestamp, the utilisation server sends the resource usage of each application to them and the applications on the services put the load on the CPU and RAM of the application using a Linux tool named stress-ng.

Changes to the Kubernetes default scheduler. Kubernetes schedules pods at the beginning of their lifetime and there is no builtin migration mechanism implemented in Kubernetes.

However, our design needs to constantly recreate services based on the users' mobility. In our design, we do the scheduling outside Kubernetes through our RL agent and other types of decision making agents and then pass the next pod placement to the Kubernetes cluster through the Kubernetes Python client API. Future work will use a more sophisticated implementation that will leverage the Kubernetes scheduler plugin to add the scheduler as a plugin to the core Kubernetes scheduler instead of doing the scheduling outside the Kubernetes cluster.

Service Migration Model. Currently there is no live migration scenario implemented in Kubernetes. Once a Kubernetes pod is scheduled on a node it cannot be moved to another node. The solution here is to stop and restart the pod in another place. If we want to move pod 1 from node 1 to node 2 we first start pod 1 in node 2. Once we are sure pod 1 is up and running in node 2 we delete its previous version from node 1. The reason for this waiting is to make sure that the user will have access to the service during the pod replacement process. In our design, we schedule pods one by one instead of using a multi-threaded version of scheduling. This is because the Kubernetes default scheduler itself does not suggest the multi-threaded movement of pods and advises scheduling them with a single scheduling queue. There have been some research efforts to implement migration mechanisms for container orchestration systems [36] and we will explore integrating these into our system in future work.

Energy saving mechanism. According to our energy consumption objective we aim to have the minimum available Kubernetes nodes in the cluster in order to be able to switch them off. Since we are using virtual machines instead of bare metal physical nodes we have not implemented a node switching on/off mechanism. For a real-world implementation, this can be achieved by draining the Kubernetes nodes to release them from the Kubernetes cluster and then switching off the physical host.

Network and Mobility. In contrast to the node and service side, the networking side is based on simulation. We have used the location of the towers in the San Francisco area [37] to generate the simulation network. We have considered co-located nodes with the stations. The nodes are then connected using a minimum spanning tree. The users are always connected to their closest base station. The users' mobility simulator is implemented using a real-world taxi traces in the San Francisco area from the cabspotting dataset [38]. We extracted the location of taxis for each five minute interval. The dataset does not contain the location of the taxis for all time intervals. To interpolate the missing entries we used the Euclidean distance and path between available points.

Training. Training the system on the real-world Kubernetes clusters is costly and time-consuming. RL agents need to be trained on the environment in several timesteps. This makes it infeasible to do the training on the real-world clusters. To solve this problem, we implemented a simulator that can mimic the dynamics of the real-world cluster during the training. We used the trained agent outside the box in the real-world cluster.

Complete System. As it can be seen in Figure 1, our design consist of three main parts. The first part is the Kubernetes cluster containing our Kubernetes nodes and services. The mobility simulator is another part that is connected to the cluster through a simulated connection that is a Python script assigning the users to the nodes. The placement of the nodes and the location of the users are passed to the controller. The controller wraps the information received from both entities into a single environment OpenAI gym [39] environment. The gym environment is then used to calculate the reward based on the current observation from the environment. This information is then passed to the RL agent to decide the next placement of pods in the nodes. We have used the rllib [40] library for the implementation of the RL agents. This placement is then passed back to the Kubernetes using the Kubernetes Python API and the pods will be moved to a new node in the cluster. All the codebase of this project is available at https://doi.org/10.5281/zenodo.7257394.

VII. Experimental Setup

We used the Google GKE service to deploy our Kubernetes cluster. Due to our computational budget, we performed the experiments on eight Kubernetes nodes. All the nodes were of e2-standard-4 type of the GKE platform with four cores and 16 GB of Memory. We used 16 stateless containerised services with the Flask app and the utilisation server explained in section VI with some constant load on them[2]. We have used pods of guaranteed QoS Kubernetes class which have equal size requests and limits. All the services are of size 250 Mb RAM and 0.125 CPU. Due to the complexity of the problem we have used constant load on the services running on the nodes. This load is generated using the utilisation server explained before. However, as we discussed in section VI, for sensitivity analysis we conducted the experiments for 16, 32, 48, users[3]. In each scenario, a service is serving one, two, and three users respectively. In our simulations, the station and nodes were co-located, therefore, we have eight stations proportional to the number of the nodes. The users move in a radius of 37.72 and 37.78 for latitude and -122.45 to -122.38 in longitude. In Figure 2 you can see the initial placement of nodes, servers, and users on the map. During the training phase of the RL algorithms, we used the simulator explained in section VI. The number of user movements available per user in the cabspotting dataset (explained in Section VI) was not sufficient for training the RL agent. Even with interpolation between the locations in the dataset location of the taxis, we ended up with 3453 locations for each taxi within five minutes intervals. Therefore, for the training phase, we generated a random user movements dataset within the same vicinity for the training with 100000 user movements. However, the final

[2]As explained before the scheduling is based on the resource request and not on the load of the containers. However, we put some load on them to more closely emulate a real-world deployment.

[3]Scaling beyond this would require a multi-agent reinforcement learning [6] instead of a single agent central scheduler and this will be explored in future work.

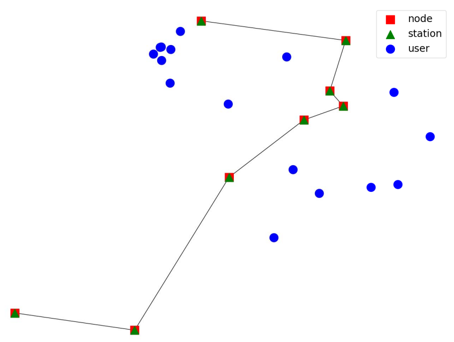

Fig. 2. Placement of the users (16 users scenario) and stations/servers on the map, we considered co-located stations and nodes (servers).

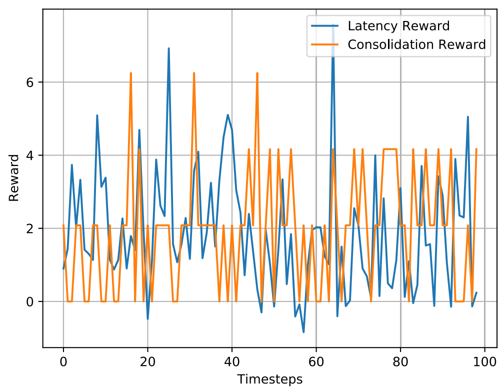

Fig. 3. Comparison of scales of the two different rewards.

test results in Figures 7, 8, 9 and 10 are generated using the real world cabspotting dataset with 3453 timesteps.

The scaling of the training rewards between the two objectives is a challenging task as the two objectives need to be scaled so that the total values of the rewards in an episode are roughly the same scale. We were able to achieve this using experimental evaluation of different scaling methods. In Figure 3 you can see that scale of the rewards within a random 100 timestep episode is almost within the same range.

RL algorithms are very sensitive to variation of hyperparameters [32], therefore using them for new problems requires lots of hyperparameter tuning. By extensive simulation, we found the following values as the optimal hyperparameters values for each of the RL algorithms. The values are presented in Table II.

In the testing phase, we averaged the results for 20 separate episodes for each of the experiments.

We compared the results of the learning algorithms with two greedy schedulers that focus on latency or energy consumption only. As there is no work with a similar setting to ours we chose two heuristic algorithms 1. Bestfit algorithm as the ground truth value for energy consumption [18] and 2. A latency greedy algorithm that moves the services to the closest

Neural Network		
# Layers	Layers size	Activation function
2	64	Linear
PG		
Train_batch_size	Gamma	Learning rate
1000	0.99	0.0003
PPO		
SGD minibatch size	train_batch_size	learning rate
128	1000	0.0003
IMPALA		
Train batch size	Gamma	Learning rate
1000	0.99	0.0003

TABLE II
HYPERPARAMETERS OF THE RL ALGORITHMS

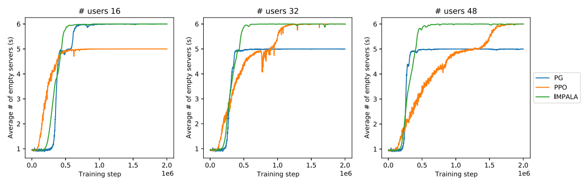

Fig. 4. Average number of empty servers during the RL algorithms training

vicinity of the users as the ground truth value for latency [6]. We compared IMPALA with two other RL algorithms that have been widely used in previous scheduling works, Vanilla Policy Gradient used in a paper with an approximately similar setting [41] and PPO which is one of the state of the art algorithms previously used in systems research and Kubernetes scheduling [42].

VIII. Results

We evaluated the result in terms of the joint energy saving and latency objectives. During the training phase of the algorithm, we observed that among the three tested RL algorithms IMPALA showed the most stable convergence. As you can see in Figure 5 and Figure 6 the green line for IMPALA has fewer variations during the training. This variation is not evident in Figure 4 as all the algorithms converge to some optimal value very quickly. IMPALA also shows a better convergence in all scenarios to the optimal energy saving objective which is 6 empty servers. There was not a consistent performance difference between the Policy gradient and PPO during the training.

For the test results, we also compared the results with the latency and energy saving heuristic algorithms. From Figure 7 we can see that the PG and IMPALA can achieve the best average latency in the network among the three tested RL algorithms. However, the difference between the achieved results in both IMPALA and PG cases is very close.

In the case of the energy saving objective, the IMPALA was able to converge to the optimal result in all network sizes. This is illustrated in Figure 8. This was not the case for the other two RL algorithms.

Analysis of the timesteps (instead of the average of the servers) of a test episode can also confirm similar results

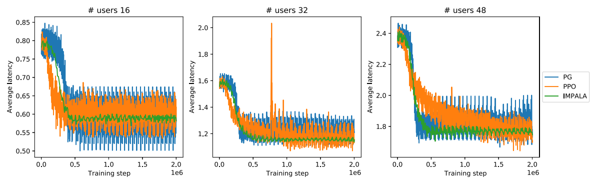
Fig. 5. Average network latency during the RL algorithms training

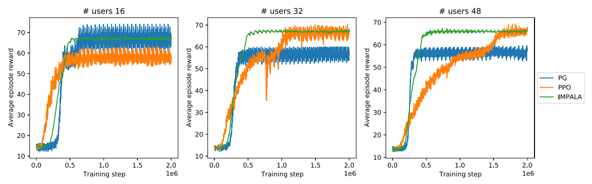
Fig. 6. Average episode rewards for different RL algorithms during the training

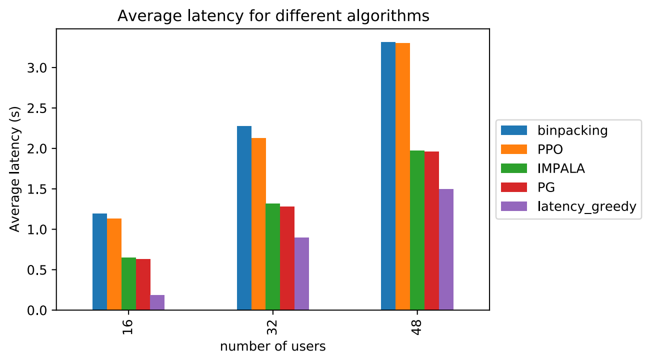
Fig. 7. Average latency of different number of users sizes for different algorithms

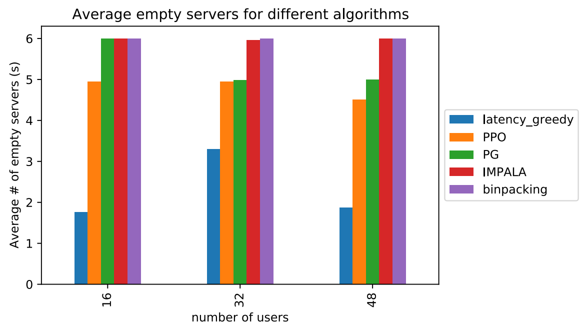
Fig. 8. Average number of empty servers of different number of users sizes for different algorithms

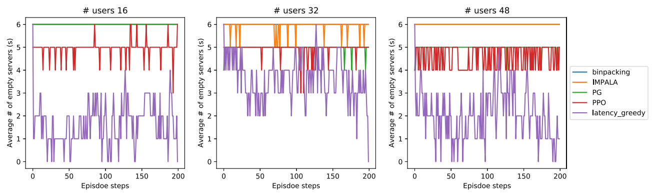
Fig. 9. Number of empty servers of different algorithms during a sample episode run

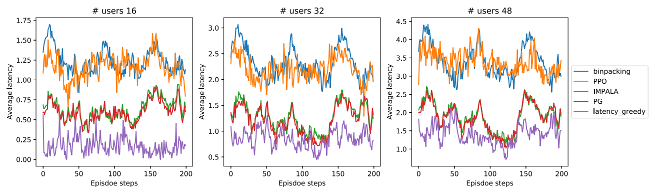
Fig. 10. Average network latency of different algorithms during a sample episode run

to averaged results of multiple episodes (previously shown test results). In Figures 9 and 10 we observe that during a sample 200 timestamp episode the ordering is the same as the averaged results. We can also see that IMPALA is able to achieve similar performance with the heuristic method that focuses on the energy saving objective only. It can achieve this result while reducing the average latency of users by 43%. This demonstrates the efficacy of the Mobile-Kube system in providing MEC services in a sustainable fashion. From these results, we observed that IMPALA is able to achieve the best result in terms of achieving a trade off between the two aforementioned objectives.

IX. Conclusions and Future Work

In this paper, we presented a deep reinforcement learning solution for reducing energy consumption in containerised clusters. Our results suggest that commonly used heuristic algorithms like bin-packing can be replaced with learning-based methods to achieve similar performance for the targeted object while also improving performance in other areas. Some directions for future works are:

Checkpointing of stateful services In this work we only considered the case the case of stateless services that could be turned on and off anytime without the need of preserving their current state. However, this is not the case for many real-world use cases. To this end, a checkpointing mechanism that could preserve the current state of the service until it is restarted can be implemented in future works.

Kubernetes full implementation Currently we have our scheduling control loop completely out of the Kubernetes cluster. Although Mobile-Kube can be deployed in a cloud environment best practice would be to place the control loop inside the Kubernetes cluster. Kubernetes custom resource and operators can be used in future works to achieve this goal.

Acknowledgments

We gratefully acknowledge Google for its support of this research through its GCP research credit program.

References

[1] "10 trends in real-world container use," https://www.datadoghq.com/container-report/.

[2] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[3] O. C. A. W. Group *et al.*, "Openfog reference architecture for fog computing. 2017," *URL: https://www. openfogconsortium. org/wp-content/uploads/OpenFog_Reference_Architecture_2 _09_17-FINAL. pdf*.

[4] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2333–2345, 2018.

[5] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[6] F. Brandherm, L. Wang, and M. Mühlhäuser, "A learning-based framework for optimizing service migration in mobile edge clouds," in *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, 2019, pp. 12–17.

[7] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, 2019.

[8] T. Bahreini and D. Grosu, "Efficient algorithms for multi-component application placement in mobile edge computing," *IEEE Transactions on Cloud Computing*, 2020.

[9] C. Liu, F. Tang, Y. Hu, K. Li, Z. Tang, and K. Li, "Distributed task migration optimization in mec by extending multi-agent deep reinforcement learning approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1603–1614, 2020.

[10] S. Ghafouri, A. A. Saleh-Bigdeli, and 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.

[11] K. Ray, A. Banerjee, and N. C. Narendra, "Proactive microservice placement and migration for mobile edge computing," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 28–41.

[12] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[13] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[14] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 416–464, 2017.

[15] Microsoft, https://blogs.microsoft.com/blog/2020/01/16/microsoft-will-be-carbon-negative-by-2030/, accessed June 2020.

[16] Amazon, https://sustainability.aboutamazon.com/, accessed June 2020.

[17] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United states data center energy usage report," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2016.

[18] F. Farahnakian, T. Pahikkala, P. Liljeberg, J. Plosila, N. T. Hieu, and H. Tenhunen, "Energy-aware vm consolidation in cloud data centers using utilization prediction model," *IEEE Transactions on Cloud Computing*, 2016.

[19] X. Li, W. Yu, R. Ruiz, and J. Zhu, "Energy-aware cloud workflow applications scheduling with geo-distributed data," *IEEE Transactions on Services Computing*, 2020.

[20] M. Xu, A. N. Toosi, and R. Buyya, "Ibrownout: an integrated approach for managing energy and brownout in container-based clouds," *IEEE Transactions on Sustainable Computing*, vol. 4, no. 1, pp. 53–66, 2018.

[21] V. De Maio, R. Brundo Uriarte, and I. Brandic, "Energy and profit-aware proof-of-stake offloading in blockchain-based vanets," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, p. 177–186.

[22] E. Ahvar, A.-C. Orgerie, and A. Lebre, "Estimating energy consumption of cloud, fog and edge computing infrastructures," *IEEE Transactions on Sustainable Computing*, 2019.

[23] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-aware application placement in mobile edge computing: A stochastic optimization approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 909–922, 2019.

[24] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on markov decision process," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1272–1288, 2019.

[25] S. S. Gill *et al.*, "Ai for next generation computing: Emerging trends and future directions," *Internet of Things*, vol. 19, p. 100514, 2022.

[26] O. Abu Oun and T. Kiss, "Job-queuing and auto-scaling in container-based cloud environments," in *10th International Workshop on Science Gateways, IWSG 2018*. CEUR Workshop Proceedings, 2019.

[27] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016, pp. 50–56.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[29] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *International Conference on Machine Learning*, 2018, pp. 1676–1684.

[30] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 197–210.

[31] N. Jay, N. H. Rotman, P. Godfrey, M. Schapira, and A. Tamar, "Internet congestion control via deep reinforcement learning," *arXiv preprint arXiv:1810.03259*, 2018.

[32] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, M. K. Shirkoohi, S. He, V. Nathan *et al.*, "Park: An open platform for learning-augmented computer systems," in *Advances in Neural Information Processing Systems*, 2019, pp. 2494–2506.

[33] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[34] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[35] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1407–1416.

[36] K. Nakashima and K. Kourai, "Migsgx: A migration mechanism for containers including sgx applications," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, 2021.

[37] "antennasearch," http://antennasearch.com/.

[38] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD dataset epfl/mobility (v. 2009-02-24)," Downloaded from https://crawdad.org/epfl/mobility/20090224, Feb. 2009.

[39] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[40] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3053–3062.

[41] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[42] S. Li, L. Wang, W. Wang, Y. Yu, and B. Li, "George: Learning to place long-lived containers in large clusters with operation constraints," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 258–272.