Received April 27, 2021, accepted May 1, 2021, date of publication May 4, 2021, date of current version May 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077550

Container Placement and Migration in Edge Computing: Concept and Scheduling Models

OMOGBAI OLEGHE
Systems Engineering Department, University of Lagos, Lagos 100213, Nigeria
e-mail: o.oleghe@ritepaklimited.com

ABSTRACT Containers are a form of software virtualization, rapidly becoming the de facto way of providing edge computing services. Research on container-based edge computing is plentiful, and this has been buoyed by the increasing demand for single digit, milliseconds latency computations. A container scheduler is part of the architecture that is used to manage and orchestrate multiple container-based applications on heterogenous computing nodes. The scheduler decides how incoming computing requests are allocated to containers, which edge nodes the containers are placed on, and where already deployed containers are migrated to. This paper aims to clarify the concept of container placement and migration in edge servers and the scheduling models that have been developed for this purpose. The study illuminates the frameworks and algorithms upon which the scheduling models are built. To convert the problem to one that can be solved using an algorithm, the container placement problem in mostly abstracted using multi-objective optimization models or graph network models. The scheduling algorithms are predominantly heuristic-based algorithms, which are able to arrive at sub-optimal solutions very quickly. There is paucity of container scheduling models that consider distributed edge computing tasks. Research in decentralized scheduling systems is gaining momentum and the future outlook is in scheduling containers for mobile edge nodes.

INDEX TERMS Algorithm, container, edge computing, migration, placement, scheduling.

I. INTRODUCTION

The number of things connected to the internet is in constant growth due to ever increasing demand for automation, artificial intelligence, augmented reality, smart homes and cities, real-time analytics, gaming and a variety of other industrial- and consumer-based applications. As a result, the volume of data being generated and the frequency and complexity of computation are also increasing, exerting pressure on cloud servers. This pressure leads to high energy usage at datacenters [1] and contributes to a reduction in quality of service (QoS) such as dropped computations, high latency, high cost of bandwidth and overloaded cloud server. Edge servers (edge nodes) have been introduced to address these and other related issues.

Edge nodes are geographically situated closer to end devices than cloud servers, as portrayed in Figure 1. The nearness of the edge node to the end device or end user is advantageous in a number of ways. Rather than send a computation from end device to cloud or transmit a response from cloud to end device, the edge server does the computation and transmission, or a part of it [2]. This service improves overall computation latency, minimizes the workload that is sent to the cloud, saves on bandwidth and enhances data privacy [3].

It is predicted that by 2025, 75% of all data will be processed outside of datacenters or cloud servers [4]. The number of edge nodes is therefore projected to increase rapidly, unlocking many opportunities for academic research in edge computing [5]–[7].

There are factors that make edge computing to be a challenging service to provide. Some of these factors are registered in Table 1. These can be classified as user-related factors, factors relating to the edge node and service-provider related factors. For instance, user computing requests vary in type, size, frequency and complexity, and users may change location during a computational request. Edge nodes in a cluster may be heterogenous in the availability of CPU, memory and energy resources. The service provider or edge network owner may be driven by one or more operational goals such as reducing energy consumption in all edge nodes or maximizing the service level agreement (SLA). In providing services at the edge, these and many other factors need to be

The associate editor coordinating the review of this manuscript and approving it for publication was Wen-Sheng Zhao.

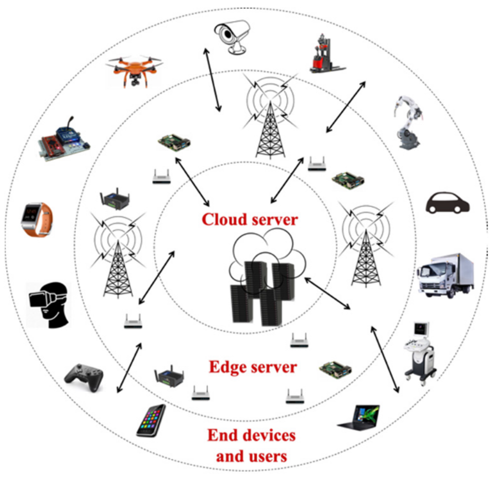

FIGURE 1. IoT architecture showing end devices, edge server and cloud server.

considered by the server manager when accepting tasks and allocating them to edge nodes.

Containers are virtualized software applications. The architecture of container-based computing platforms enables containers to be deployed, terminated, replicated, recovered and migrated in milliseconds. For server managers, these features help improve system flexibility and computing scalability [8]. These and many other features encourage the use of container-based applications as the platform for enabling cloud and edge computing.

Container orchestration is the term used for managing multiple containers on different computing nodes. Although container orchestration is at an advanced stage in cloud servers, it is still in the infancy stage in edge computing [9]. One of the key components of container orchestration is the scheduler, responsible for deciding which container application is placed on which edge node. From a service provider standpoint, the scheduler decides how the different incoming computing requests are mapped to containers, and how containers are mapped to heterogenous edge nodes, to service the request. These mappings attempt to fulfill a set of objectives within given operational constraints [10]–[13]. The scheduling decision is a complex one. In fact, the container placement problem in edge computing has been described as NP-hard to solve [14]–[16].

Researchers have promoted an assortment of scheduling models to address a variety of container placement problems in edge computing. This study focuses on the frameworks and algorithms upon which the scheduling models are built. The aim is to illuminate the key methods that have been used to formulate the container placement problem in edge computing as well as the types of algorithms used in arriving at an optimal placement solution. The resultant findings can be used as a reference point for building container scheduling models for edge servers.

Some research has been done in related fields, see for example the studies reported in [17]–[20], but these have not been in the context of containers. Others have reviewed the use of containers from a general perspective, without special emphasis on scheduling of containerized edge computing, see for example studies in [21], [22]. Ahmad *et al.* [23] report a survey of scheduling techniques in edge computing. In their review, they present the scheduling methods that have been used to fulfil container-based edge computing. The current study is unique as it highlights the frameworks and algorithms upon which the scheduling models are built. The analysis of the framework describes how the scheduling problem is converted to one that can be solved analytically. The algorithms are used in solving the converted problem in a timely and efficient manner for the scheduler. The frameworks and algorithms are the linchpins to scheduling models.

TABLE 1. Factors that challenge the provision of edge computing.

Factor type	Factors
User-related	mobility, request size, request type, request frequency, location
Edge node-related	heterogeneity, resource and availability, location
Service provider-related	load balancing, cost, bandwidth, energy, SLA

The scope of this study is container placement and migration, from the standpoint of the provider of edge computing services. The main contributions within the study scope are outlined as follows: 1) clarifies the concepts of container-based edge computing, container placement and migration in edge computing, and container scheduling for edge computing; 2) reviews the Kubernetes and Docker Swarm schedulers as well as the extensions that have been developed for them; 3) underscores the frameworks and algorithms that have been used to bring about the efficient scheduling of container-based services in edge computing.

The remainder of this article is organized as follows: section II provides an overview of the concept of containerized edge computing; section III outlines container placement and migration; section IV presents an analysis of Kubernetes and Docker Swarm schedulers in edge computing, including the improvements that have been developed for them; section V reports on the frameworks and algorithms that have been used to develop the assortment of scheduling models; section VI discusses the key findings; section VII is the conclusion.

II. CONTAINER-BASED EDGE COMPUTING: CONCEPTS

Understanding the concept of edge computing and containerized applications is crucial to understanding why and how container placement decisions are made. In this section, the concept of edge nodes and edge computing are clarified with respect to container placement scheduling. Application examples are presented to show the variety of computing requests that are fulfilled on edge servers.

A. EDGE NODES AND EDGE COMPUTING

The characteristics of edge nodes determine how edge computing services are managed and delivered. Edge nodes can be described as computing devices equipped with network connectivity to enable them communicate with end devices, other edge nodes and the cloud. Some characteristics have been defined for edge nodes such as: resource (CPU, memory, disk, power) constrained, low cost, low communication bandwidth, low security [24], [25], sparsely distributed in a network or geographic region [24] and diverse computing configurations [26]. They have limited service range and so need to be optimally located to serve end users. The distinguishing feature for an edge node is that it brings computing closer to the end user [27]; computing that would normally have been fulfilled on the cloud. Any single device with computing power that serves this purpose can be an edge node, whether a single board computer or a desktop computer, be it stationary or mobile [12]. A cluster comprises a group of more than one edge node, so defined for ease of managing from a centralized location. In a cluster the distance between edge nodes can range from a few meters to a few kilometers. The inter-edge node distance cannot be much further due to the network range limitations of most edge devices.

Edge computing, sometimes called fog computing [22], [28]–[30] can be defined as the data computation that is deployed and fulfilled on edge nodes rather than on the cloud. The main types of edge computing tasks have been machine learning and deep learning [31], augmented reality with object recognition [32], [33] and network virtualization [34], [35]. Some computations such as deep learning and augmented reality are computationally expensive i.e. they demand much CPU and memory to run. A number of devices used as edge node, for instance Raspberry Pi and other single board computers, are resource constrained to undertake computationally expensive tasks. This motivates the dividing up of such computations across two or more edge nodes [31], in a distributed or parallel computing manner. The heterogeneity of edge nodes and the variety of edge computing tasks, are factors (see Table 1) that container scheduling models need to consider when scheduling the placement of container applications on edge nodes [36].

B. CONTAINER-BASED VIRTUALIZATION

Application virtualization can be fulfilled using virtual machines or containers. With virtual machines, a virtual operating system is hosted on the disk [37]. In containers, the memory allocated to running applications is run on top of a hosts operating system's kernel [38]. A container can be described as a live application environment that has occupied a portion of computing resources of a host machine [39].

Researchers have assessed the performance of edge computing on the basis of containers, virtual machines and bare metal native host. Empirical results show that rate of task execution is hardly diminished using containers as against using native host, unlike using a virtual machine [37], [40], even for deep learning computations [41]. Joy [42] found that the time taken by a Linux container to scale and process a service request is significantly less than the time taken to scale a new virtual machine to handle the same request.

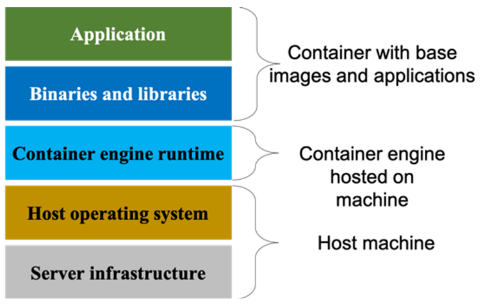

FIGURE 2. Architecture for container virtualization.

The virtual machine took as long as three minutes while the container took just eight seconds. Joy [42] also found that more applications can be deployed on a host machine when using containers than when using virtual machines. Containers were found to have low storage footprint due to the sharing of the same image [43]. Others have compared the performance of various container engines. Manninen *et al.* [44] tested an I/O intensive python application for a control and monitoring system, using Docker, Balena, Linux and Flatpack containers. They focused on memory, CPU and power consumption. They found that Docker was the most resource-hungry platform, with Flatpack and Linux using the least resources. Park *et al.* [45] develop a model that shares execution environment containers across multiple application containers, which helps to achieve significant reductions of the resource pressure compared to using Docker-based monolithic containers.

A container is a complete runtime environment comprising an application with its dependencies. The dependencies may include an operating system (such as Ubuntu), codes, libraries, binaries and configuration files, that are needed to run the application. The application together with its dependencies are all together encased in a single package, called a container [46]. Figure 2 depicts how a container is stacked in a host machine. For a container to function, its container engine, needs to be hosted on the machine. Examples of container engines are LNX, Docker, Mesos, OpenVZ and Containerd. The Docker engine for example comprises a Docker daemon, a REST API and CLI, the whole of which take up roughly 1.5Gb of host machine disk space.

A container is split into base image layer and application layer (see Figure 2). Examples of base images include Ubuntu and Alpine. Some base images such as Red Hat include an operating system, language run times (NodeJS, python and PHP) and a set of YUM repositories [47]. Base images can be purpose-built such as in Red Hat, but many container applications need only a base operating system such as Ubuntu and Alpine. The base image is a read-only layer in the file stack of the container.

The application itself (read and writeable layer) may include pre-built, purpose built, opensource or commercial microservices. Some opensource microservices have been built for TensorFlow, MongoDB, Grafana and Eclipse Mosquito. Combining multiple microservices in a single container is sometimes inevitable when fulfilling complex and computationally demanding tasks [13], [48]. However, this increases the container file size. Containerizing microservices has been found to be better than running multiple microservices in one container [10]. Although this increases the complexity of orchestration, it boosts the container boot up time because all the microservices can be booted simultaneously.

A container is a running instance of the base image together with the microservice(s) or apps. A container application can only run when its base image is present on a host machine. One base image (parent image) can be used to deploy different applications, if the applications have similar dependencies. Figure 3 portrays how different applications can use the same base image. In the depiction, the Grafana and MQTT applications share the same base image i.e. Ubuntu. The python applications also share the same base image of Ubuntu which are all stacked on TensorFlow. Where a base image is already deployed, the container takes a shorter time to boot up, than where the base image is not present. This is possible because the container is able to recognize that its base image exists in the host container engine. The container boots up with only the microservice(s), rather than boot up a base image first, which takes more time to deploy due to the large file size of most base images. This is what makes containers to be rapidly deployed [49], and is sometimes a factor to consider when deciding where to place containers in edge nodes. For instance, to deploy a python-based application that relies on TensorFlow, the scheduler can prioritize nodes that already have the Ubuntu and TensorFlow images.

C. CONTAINER APPLICATIONS IN EDGE COMPUTING

The scope of container applications cuts across every industry [50]. In the literature, the use of container-based edge computing was found in: a civil construction application [51], a crane maneuvering application to avoid accidents with construction workers [52], monitoring patient fall detection [29]; human activity recognition [53]; autonomous vehicles [33]; real-time video processing [3], [41]; real-time process control [54], [55]; intelligent farming [56]; factory system software management [57] and modular cyber-physical system [58]. In [59], authors present the implementation of a container-based MQTT broker. To avoid overload in any broker, a load balancer uses an algorithm to route new client requests to the least loaded broker. Authors in [60] parallelize the training of a machine learning regression model using Linux containers deployed on participating edge nodes. Similarly, in [31], the authors distribute a container-based TensorFlow deep learning task among several Raspberry Pi devices.

The virtualization of network functions has been enhanced with the use of containers. The authors in [35] present an approach to launch virtual network functions on demand on edge nodes. Container-based network functions can be used to quickly re-establish lost network service by uploading the network functions as containerized applications.

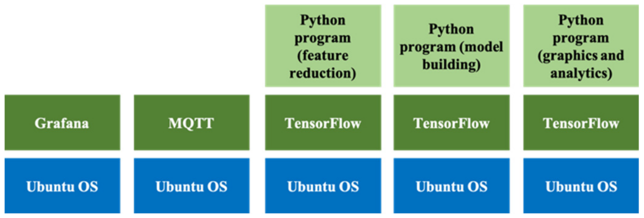
Python program (feature reduction)
Python program (model building)
Python program (graphics and analytics)
Grafana
MQTT
TensorFlow
TensorFlow
TensorFlow
Ubuntu OS
Ubuntu OS
Ubuntu OS
Ubuntu OS
Ubuntu OS

FIGURE 3. Container stack with base image and microservice application.

Such network functions include virtual private network, diagnostics/ monitoring functions, encryption and firewall.

Edge computing tasks vary in type. The container scheduler needs to take cognizance of this when deciding how to accept and allocate requests to containers, and map containers to edge nodes. For instance, an incoming object recognition task may not be schedulable due to unavailability of an edge node. The scheduler has the option of either dropping this request or allocating it to a cloud-based server. The scheduler can also decide to migrate an already running computation to another edge node so that the incoming request can be accommodated in the server.

III. CONTAINER PLACEMENT AND MIGRATION IN EDGE COMPUTING

In addition to the varying nature of computing tasks received at an edge server, there is randomness in when they are received. Tasks may be received intermittently or continuously. In edge computing, a container is not deployed unless there is a request for its services. A container by definition is a running instance of an application. As such, a container can service only one computing request since an already running application cannot be used to service a new incoming request. However, multiple containers can be deployed to service a single request, but two different requests cannot be serviced by the same container.

There are two distinct types of container placement approaches namely queuing and concurrent [16], [61]. The queuing approach can be abstracted as a first-in-first-out or priority-based method [62], [63], where the container placement decision is made on a container-by-container basis. A batch and process concept can be used to describe the concurrent approach where computing requests are first collated and then placement decision is made. Figure 4 is used to depict the two approaches. As the example in Figure 4a shows, container 4 cannot be placed on any of the edge nodes after containers 1, 2 and 3 have been placed one after the other. In the concurrent approach, Figure 4b, container placement is optimized.

Taking decisions on a container-by-container basis can lead to load imbalance, necessitating the need to relocate already running services to create a balanced cluster. In the queuing approach, a global optimal decision is difficult to achieve [62], because the immediate placement decision is taken only for the first container on the queue, without consideration for subsequent containers on the queue. However,

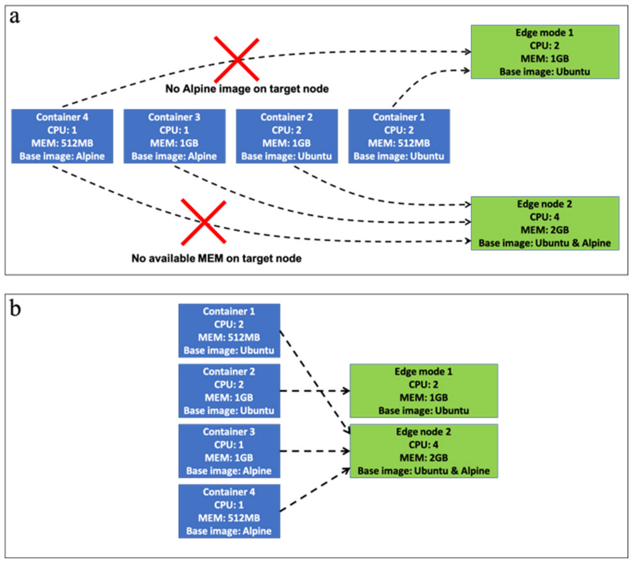

FIGURE 4. Container placement approaches: a) queue-based placement; b) concurrent container placement.

it is possible to set dispatching rules for container placement, if there is prior knowledge of the characteristics of all incoming requests. A machine learning-based predictive model is used to predict incoming request to supplement such an approach [64], [65].

A concurrent approach is able to achieve a global optimal schedule, through the batch and process method. The drawback with this method is that batching time can add to placement delay if incoming requests are intermittent, because the allocation of tasks waits for a specified time to collate multiple tasks [66]. For a continuous stream of incoming requests, a concurrent approach can be used, while for intermittent requests, the container-by-container approach can be used.

Container placement also encompasses container migration i.e. relocating an already running service to another edge node. One reason for migrating a running service may be because an end device/user has changed location such as an autonomous vehicle. Another reason for migrating container services is to balance the workload within the cluster of edge nodes, so that no edge node is overloaded, while others are underloaded. A third reason is that an edge node can suddenly become unavailable, such as an unexpected shutdown or a planned maintenance and a running application needs to be migrated to continue an already started computation.

When a container is migrated between edge nodes, it is the base image and microservices (see Figure 3) that are migrated. Computing can equally be migrated to the cloud from an edge server, in a situation where there is no suitable edge node to transfer services to, and then back to an edge node when one becomes available [67]. Where an edge node cannot complete a computation due to limited resources, part of that computation can be moved to the cloud [10]. Containers are used in cloud servers and so seamless migration is possible between edge node and cloud server. For these reasons, containers facilitate efficient service handoff or transfer in mobile and migrating edge services.

Container migration is of two types: cold and warm. In the cold migration, the container(s) and base images are together transferred to another node, in a single step process. In warm migration, images are first replicated on the target node, ahead of the migration, in a timely manner. Afterwards the running applications are frozen, saved on the source node disk, and then offloaded to the target node, on top of the already deployed images, making a multi-step process [68]. Containerized applications with base images that are large in file size can benefit from such migration. When the base image is already deployed, i.e. pre-cached in the edge node, deploying only the container(s) takes less time than deploying both image and container at the same time, as it is done in cold migration. This way, the temporary loss of service during migration is minimized than in a cold migration. Caching methods have been comprehensively investigated in [65].

Some applications have very strict latency requirements such as autonomous driving, which can benefit from warm container migration. Authors in [33], [68] tested the warm approach on a 2GB container-based facial recognition application. They were able to reduce container downtime during migration from 3,200 seconds to 49 seconds on a 5Mbps bandwidth. Authors in [69] propose a similar approach that proactively deploys multiple instances of the client runtime application in neighboring edge nodes. Service replication is through pre-cached images.

There are variants of the warm migration, such as pre-copy and post-copy, with conditions best suited for using them [70], [71]. Warm caching is good in terms of reduced latency, but it uses up limited resources such as memory [72]. A scheduler has to consider the cold and warm migration options when deciding where to place a container. Kubernetes scheduler uses the predicates and priorities scheduling policies for this purpose see [73].

Container migration whether cold or warm can be of two types namely stateless and stateful [74]. Stateless migration is the easier of the two. In stateless migration, the previous state of the runtime container is not transferred, rather a new container image is replicated in the target node. If stateless migration is to be used for an already running container application, it means the computation will be terminated and restarted afresh, resulting in loss of already done computation. This results in duplicated computation. The total computation time will increase if it is a high workload task such as video analytics and if the computation has significantly progressed in the source node before termination. Stateless container migration has been demonstrated in a computationally expensive object detection classification task [3].

In stateful migration, the state of the application prior to migrating from the source node is saved and transferred to the target node. The runtime container image at source after offloading is the same at target after handoff. Checkpoint/Restore in Userspace (CRIU) is a well-known

Linux-based software used in implementing stateful migration [75]. The software freezes a running application and checkpoints (saves) this state to disk as a collection of files. The application is restored and continued on the same device or on another device from its frozen state. The application state at restoration is exactly the state at freezing [76]. A scheduler needs to factor in this process, when deciding where to place computations that are already being serviced.

IV. KUBERNETES AND DOCKER SWARM CONTAINER SCHEDULERS FOR EDGE COMPUTING

Kubernetes is the prevalent container orchestration tool in both cloud and edge computing. Developed by Google, it is maintained and supported by a large opensource community. Kubernetes is integrated in many cloud computing services such as Google Cloud, Amazon Web Service, Red Hat and Microsoft Azure. Kubernetes automates the deployment, scaling and management of container-based applications. Details of Kubernetes can be found in [73].

Kubernetes scheduler is one of the main components in the Kubernetes architecture. The scheduler decides the placement of containers (also known as pods) unto edge nodes. It relies on a greedy multi-criteria decision-making framework. In its simplest form, nodes are labelled using a weighted sum of pre-determined metrics such as CPU and memory. The scheduler considers other factors relating to nodes and tasks, such as data locality, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines [73]. The Kubernetes scheduler relies on these metrics and factors when deciding where to place containers. Kubernetes allows custom metrics and factors to be built into the scheduler. Pods are able to communicate their metrics to the Kubernetes central server also called the control plane. This way the resource utilization in each edge node is monitored at heartbeat intervals. The scheduler relies on this information amongst others, to schedule pods in such a way that the workload is evenly distributed and other requirements are fulfilled. The Kubernetes scheduler taint/toleration and predicates/priorities mechanisms are also used to restrict which pod is placed on an edge node [73]. Chima Ogbuachi *et al.* [77] give a good account of the Kubernetes scheduler backend functions.

Kubernetes has largely been used for cloud-based computing. Many of the requirements of edge computing are different from those of cloud computing. For instance, resource limitation is more evident in edge nodes than in cloud-based nodes. Edge nodes are more geographically spread than cloud servers, and Kubernetes does not consider the topology of the edge network [2], [9]. Heterogeneity of nodes is more pronounced in edge servers than in cloud servers. The latency requirements are more stringent for edge computing than in cloud computing. In addition to these, Kubernetes default scheduler uses the queuing model, which has some limitations.

Extensions have been proposed to enhance the Kubernetes scheduler component and mitigate some of its limitations in edge servers. Authors in [36] introduce Skippy, a kubernetes plugin scheduler. Their model aims to address some limitations of Kubernetes such as non-consideration of the tradeoff between data movement and computation movement, non-consideration of the inter-edge node bandwidth and non-consideration of the proximity of databases. The scheduler makes heuristic trade-offs between data and computation movement, and considers workload-specific compute requirements such as GPU acceleration. The scheduler uses a python-based discrete event simulation tool to find the values of weighs for the priority functions. In their model, the scheduler prioritizes:

1. nodes (whether cloud or edge), where the computation happens more quickly.
2. nodes where the network bandwidth is high.
3. nodes with high computing resources such as a GPU.

Kubernetes scheduler by default is run from a centralized node, also known as the master node, which runs the control plane API server. The master node manages and controls the worker nodes. The workload at a master node increases as the number of worker nodes increases. This can create problems for the master node such as high latency, high energy consumption, high resource utilization and frequent maintenance. Casquero *et al.* [78] describe a custom scheduler for Kubernetes orchestrator that pushes the scheduling decision to the worker nodes. Kubernetes node filtering and ranking functions, predicates and priorities respectively, are moved from the master node to the edge nodes. Their model is able to schedule faster than the centralized scheduling model used by the default scheduler in Kubernetes.

Haja *et al.* [79] design a custom Kubernetes scheduler that makes decisions based on applications' delay constraints and edge reliability. In their model, latency measuring pods are launched on nodes. These pods ping each other periodically, and record the message round-trip time as a node-to-node latency measurement. The node-to-node latency measurements are cached as key value pairs and used by the scheduler to match computing tasks with their respective latency requirements.

Kaur *et al.* [14] extend the functionalities of the Kubernetes scheduler by considering the carbon footprints and energy consumption at the edge network as part of the decision variables. They formulate the requirements as a multi-objective optimization problem, solved as an integer linear programming problem. A model proposed by Chima Ogbuachi *et al.* [77] aims to take the physical, operational, network and software states into consideration. The authors apply a hybrid scheduling model combining both the queuing model and the concurrent model. In a study by Fahs and Pierre [80], the Kubernetes scheduler is modified to prioritize the placement of pods in edge nodes that are located close to the main sources of network traffic. By so doing, the network latency between end device and edge node is considerably reduced. More recently, Han *et al.* [81] introduce KaiS, a Kubernetes-oriented scheduler. The KaiS scheduler relies on each edge node taking the dispatch decision. They use a Multi-Agent Deep Reinforcement Learning-based scheduling model that places a dispatching agent at each edge access point. The edge access point receives requests and dispatches to edge nodes.

Kubernetes allows custom scheduling logic and decision variables to be added to the default model in a variety of ways [9], but this can create additional computation overhead for the scheduler [77]. A custom scheduler can be built and implemented outside of Kubernetes control and made to interface with Kubernetes through an external module [77]. The external custom scheduler runs side-by-side the default Kubernetes scheduler and either of the two schedulers is invoked depending on the requirement.

Docker Swarm is another popular container orchestration tool, native to the Docker engine. It implements two simple scheduling strategies namely random and spread [28]. In a random strategy, containers are placed randomly on nodes, while a spread strategy attempts to balance the load, and so places new containers on nodes with the least workload. The spread strategy gives a better latency than the random strategy [28]. Docker Swarm scheduler is advantageous due to its simplicity, but is not as robust as Kubernetes for most real-world use cases.

As with Kubernetes, some researchers have proposed models to enhance Docker Swarm for edge servers. Authors in [28] extend the Docker Swarm functionalities. The scheduling algorithm is based on the Dijkstra's algorithm which calculates the latencies between the end users and the edge nodes. Another algorithm selects the most adequate edge node to place a container service, such that latency is minimized. In [82], the authors develop a model which scans all container deployment requests and prioritizes them according to metrics defined by the orchestrator. A component in the model periodically probes the edge nodes to get the relative latency between cluster nodes. It stores the information as a table, sorting the nodes in terms of relative latency. The scheduler allocates containers on the basis of latency and distance of user to edge node. Mendes [83] extend the functionalities of Docker Swarm to make it more flexible to overbooking. Overbooking is allocating more resources beyond the nominal capacity of the edge node. The author designed an algorithm that uses the energy efficiency levels at the nodes to schedule container placement. In the scheduling model three levels of resource utilization are defined namely: low, desired and degradation energy efficiency. Their model utilizes a host registry that lists, in descending order, the total resource utilization of each host in each level. Priority scheduling is given to the first elements (nodes) in the list of low energy efficiency nodes, so that those nodes can move to the desired level of energy efficiency.

There are other container orchestration tools such as Marathon by Apache Mesos and Cloudify [21]. Within the literature on scheduling models for container placement in edge computing, there is paucity of reported use cases for these and other orchestration tools.

V. CONTAINER PLACEMENT SCHEDULING FRAMEWORKS AND ALGORITHMS FOR EDGE COMPUTING

The container placement problem in general is a decision-making problem. It is how the edge server scheduler can optimally allocate incoming computing requests and relocate already running computations, given a set of user requirements, operational goals and system constraints. The decision is for multi-container placement in a multi-node cluster.

Container placement scheduling models are continually being advanced for edge computing. The focus of this study is not on the functionalities of scheduling models, since functionality is influenced by the decision factors relating to the specific use case. This study focuses on the frameworks that have been used in building the scheduling models. The frameworks explain how the container placement problem is formulated i.e. how to convert the problem to one that is solvable, analytically and logically. The problem formulation determines the logic that is inputted into the algorithm, as well as the type of algorithm that is used, because the algorithm is used to solve the formulated problem. In addition to the framework, this section reviews the types of algorithms that have been used to solve the container placement problem, such as heuristic, metaheuristic, graph-based and deep reinforcement learning.

A. SCHEDULING FRAMEWORKS FOR CONTAINER PLACEMENT PROBLEM IN EDGE COMPUTING

The container placement problem can be formulated in different ways. The Kubernetes scheduler for example is based on a greedy multi-criteria decision-making framework. This section presents the scheduling frameworks that have been used to model the variety of container placement problems in edge computing.

1) OPTIMIZATION-BASED FRAMEOWRKS

Optimization modelling is a well-known method of modelling complex problems, using analytical functions. The container placement and migration problem in edge computing can be modelled using optimization modelling, as it consists of a set of decision variables, decision constraints, objective functions and model assumptions [84]–[86]. In the container placement problem in edge computing, the objective function relates to the goals such as minimizing latency [84], minimizing overall cost of providing edge services [12], [39], minimizing total energy used by the server [86] and minimizing load imbalance [39]. Satisfying SLA is a maximization objective [10], [33]. Many of the problems tend towards a joint optimization problem, see examples in [14], [39], [71], [84]–[86]. This addressed the tradeoffs that may exist while attempting to fulfil one objective over another.

The decision variables are the scheduling decisions to be taken, such as which computing tasks to accept [10], [87], what containers will the task be assigned to [87], what edge nodes will the containers be deployed [33] or migrated [39]. Constraints in optimization problems can be described as finite resources that the decision maker has to consider when taking the decision. Constraints for an optimization problem are case dependent, i.e. each unique problem has its own distinct set of constraints. Authors in [10] propose to maximize system utility, using a set of constraints that guarantees that the sum of the resources needed to deploy and run the container on the edge node is less than the total amount of available resources on the node. In [11] the authors propose a data block placement and task scheduling joint optimization model. In their model one of the constraints forces each task to be processed on only one container. Others have modelled energy, CPU and latency-related constraints [12], [13] and bandwidth constraints [12].

The modelling of the container placement problem using an optimization-based framework permits the problem to be converted to one that is mathematically solvable. The scheduling algorithm works to find a solution that fulfils the objective function. By so doing, it arrives at an optimal or near-optimal solution.

There are some drawbacks with using optimization models generally. The model designer has to be well grounded in integer, linear and non-linear program modelling. The complexity of the optimization problem, and the time taken to solve it, is a factor of the type and number of constraints. Non-linear constraints add complexity. For this reason, most constraints used in the optimization model are either linear or binary [88], and the number of constraints is normally below five. Another drawback is with the use of assumptions, which can reduce the real-world efficacy of the eventual model.

2) MULTIDIMENSIONAL KNAPSACK

The Knapsack model can be used in situations where there are more items than can be accommodated in a given space. The knapsack problem can be described within the context of containers in edge computing. Suppose each edge node is a knapsack with CPU capacity and each container is an item with CPU requirement to be placed on the edge node. The knapsack problem exists when there are more container applications than the edge node CPU capacity can accommodate.

Assume there are n container applications, w is the CPU requirement of each application and c is the CPU capacity of the edge node. Suppose also that v is the degree of value of the application. For instance, if the owner of the edge node seeks to maximize SLA on latency, an object recognition request from an autonomous vehicle will hold more value than a request to filter data for onward transfer to a cloud-based database. The objective of the scheduler is to select, at every time step, a subset of requests to place in the edge node, such that the overall value is maximized within the constraint c. This can be modelled as a knapsack problem using Equations 1-3 [89]:

$$\boldsymbol{max} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{v_i x_i} \tag{1}$$

$$\text{subject to: } \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{w_i x_i} \leq \boldsymbol{c} \tag{2}$$

$$\boldsymbol{x_i} \in \{\boldsymbol{0}, \boldsymbol{1}\} \quad \forall\, \boldsymbol{i} \tag{3}$$

The binary decision variable x_i is introduced for each container application, where $x_i = 1$ if the i-th request is selected and placed on the edge node, otherwise $x_i = 0$ (hence the name 0-1 knapsack problem). The knapsack problem has been proven to be NP-hard [89]–[91].

In the real world, there are multiple edge nodes as well as multiple resources e.g. CPU, memory, disk and energy. In such a scenario, the problem becomes a multidimensional knapsack problem (MDKP), formulated as follows:

Assume there are m numbers of edge nodes with resource capacities c_j for $j = 1, \ldots, m$. The binary decision variable can be represented as $xi_{,j} = 1$ to represent that container i is selected and placed into edge node j, $xi_{,j} = 0$ otherwise. The MDKP can be formulated as Equations 4-7.

$$\boldsymbol{max} \sum_{\boldsymbol{j=1}}^{\boldsymbol{m}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{v_i x_{i,j}} \tag{4}$$

$$\text{subject to: } \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{w_i x_{i,j}} \leq \boldsymbol{c} \quad \forall\, \boldsymbol{j} \tag{5}$$

$$\sum_{\boldsymbol{j=1}}^{\boldsymbol{m}} \boldsymbol{x_{i,j}} \leq 1 \quad \forall\, \boldsymbol{i} \tag{6}$$

$$\boldsymbol{x_{i,j}} \in \{\boldsymbol{1,0}\} \quad \forall\, \boldsymbol{i,j}. \tag{7}$$

Authors in [11] model the placement of data block containers on edge nodes as a 0-1 MDKP. Authors in [33] modelled each edge node as a knapsack with constrained CPU and memory, while each application is considered as an item to be placed in each knapsack. In [10] and [92], the authors model the selection of tasks to accept and those to place on containers as a multidimensional knapsack problem.

MDKP is typically solved using dynamic programming and branch-and-bound techniques [89]. These techniques are hampered in large scale MDKPs, which are better handled by algorithms such as Tabu search [11].

3) MARKOV DECISION PROCESS

Markov Decision Process (MDP) provides a mathematical framework for modeling a system as a set of states and actions to be taken given the current state. MDP framework seeks to find the immediate optimal actions, given the current state situation, while considering the future outcomes. Subsequently, the key elements in MDP are discussed in relation to container placement problem in edge computing.

In MDPs, an agent can be described as the decision maker, for example the container scheduling agent [81]. In an MDP, the state space is the set of situational conditions that initiates a decision to be made [93], such as the changing distance between an end device and edge node, which triggers a container migration [94]. A state can also be the current workload [64], or the queue of computing requests [63].

An MDP also includes a set of actions that can be performed by the agent, when moving from one state space to another, for instance to migrate or not to migrate a container [88]. Selection of an action is through exploration, where the agent randomly selects an action [95], through exploitation, where agents take actions based on previous actions taken [96], or through a balancing of both strategies [95]. Rewards are assigned for taking an action in a state. In the context of container-based edge computing, the reward can be linked to the objective function, for instance to minimize total system cost over time [94], [97]. In [85], the authors explain reward succinctly as computation completion time. The shorter the time, the higher the reward, and any computation that exceeds the SLA time limit is punished with a negative reward. Accumulated rewards are calculated at each time step or action and stored as Q-values, to be used with an exploitation action selection approach. An agent in MDP tries to maximize the accumulated rewards it receives from each action.

In MDP, a policy is a decision rule used at every decision time step [93]. It is also a sequence of decision rules to be used at all decision epochs [94]. In [88], the author gives examples of such decision rules as whether to always migrate, never migrate or infrequently migrate containers. A policy is a function of the current state. The goal of an MDP is to find the policy at each state that maximizes the expected reward of executing that policy, given the current state situation.

Discount factors are used to balance the trade-off between immediate and future rewards [98], i.e. it controls the weight given (between 0 and 1) to short-term and long-term rewards. Otherwise, reward accumulation becomes unending. A discount factor closer to 0, implies that immediate rewards are more important, and rewards obtained later are discounted more than rewards obtained earlier. As an example, if computation latency is to be minimized, then by setting a discount factor $\longrightarrow$ 0, an MDP-based algorithm will minimize the instantaneous latency. By setting a discount factor $\longrightarrow$ 1, the algorithm will minimize the long-run system latency, without considering the short-term consequences. The setting of discount factor is application dependent [98]. However, some guidelines exist, for instance, if the service duration is going to be long, the discount factor can be set closer to 1, since a long-term horizon is the case [88]. The discount factor values are plugged into the computation for the accumulated rewards.

Han *et al.* [81] investigated how edge access points can optimally service computing requests that are received from multiple edge nodes. Containers are dispatched from the edge access points to the edge nodes. They modelled a multi-agent deep reinforcement learning scheduling model using MDP. The edge access point is the agent. The state space in the formulation consists of: the service type and delay requirement of the current dispatching request, (ii) the queue information of requests awaiting dispatch at the edge access point, (iii) the queue information of unprocessed requests at the edge node, (iv) the available CPU, memory and storage resources at the edge node, (v) the edge node number label and (vi) the measured network latency between the edge access point and the cloud. The individual action space of the edge access point agent specifies where the received request can be dispatched to. Each agent attempts to improve the long-term throughput while ensuring the load balancing of the edge network.

MDP framework is the basis for Deep Reinforcement Learning-based algorithms. These are amongst the most efficient algorithms. By framing the placement problem using MDP, the scheduler is being set up for a deep reinforcement learning-based model. MDP has a lot of moving parts (elements), which adds complexity to the modelling process.

4) LYAPUNOV OPTIMIZATION

Container placement can be viewed as a queuing system, where containers are queued or buffered before placement. This type of system can be modelled using the queuing-aware Lyapunov optimization framework. When dealing with network queues, the Lyapunov optimization framework has proven to be beneficial. It is used to achieve optimal control of the system by introducing dynamic coefficients [12]. The state of a system at an instant in time can be described using a non-negative multidimensional function, called a Lyapunov function. The function is defined in such a way that it grows when the system moves towards undesirable and unstable states. The objective is to make the Lyapunov function drift towards zero (stability). A typical goal is to stabilize all network queues while optimizing some performance objective(s) [63], [86], such as minimizing average energy [86]. The Lyapunov function can be described as follows:

Suppose an edge network evolves in discrete time $t \in [1,\ldots,n]$. The task accumulation at time slot t given by the Lyapunov function is given by Equation 8.

$$\boldsymbol{L(t) = \frac{1}{2}\sum_{i=1}^{n} Q_i(t)^2} \tag{8}$$

The Lyapunov drift is the difference in Lyapunov function values at two consecutive time slots [96] and is given by Equation 9.

$$\Delta L(t) = L(t+1) - L(t) \tag{9}$$

A drift closer to zero indicates a more stable queue, which is the ideal case. To quantify the effect of the drift on the system, such as system cost or energy, a non-negative control coefficient V is introduced [12]. If the scheduler is to stabilize the task queue while minimizing the time average system cost C (container placement cost), the drift-plus-penalty expression $f(t)$ is defined as Equation 10:

$$f(t) = \Delta L(t) + V \cdot C(t) \tag{10}$$

The V parameter can be chosen such that the time averaged $C(t)$ is arbitrarily close to optimal. The larger the value of V, the greater the emphasis on cost [96]. The problem is then reduced to minimizing the upper bound of the drift plus penalty function [96]. By controlling the $f(t)$ at each time step, the scheduler can control the queue length while minimizing system cost.

Authors in [63] build a dynamic service migration and workload scheduling model. Their scheduler is able to decide where services should be migrated in response to user mobility and demand variation. They model the problem as a sequential decision making MDP. They use Lyapunov optimization approach to convert the constrained MDP into a sequence of unconstrained stochastic shortest path problems, solved over consecutive renewal time frames. Some researchers have investigated a system whereby parked vehicles with computing capabilities can be added to a network of edge nodes [86]. They studied how computing services can be migrated from fixed road side edge servers to the parked vehicles. They first model the container scheduling problem as an optimization problem. The Lyapunov optimization function is then used to reduce the optimization problem to a per-slot optimization problem. This way only the current time slot information is needed. In [12], the authors studied service migration for vehicular edge computing. They formulate and solve a cost-minimization container deployment problem, using the Lyapunov function. The Lyapunov-based algorithm allowed the deployment decision to be made greedily, based simply on the current system states without considering the future knowledge of stochastic factors. Authors in [99] design a service placement scheduling model for cost-efficient mobile edge computing. They propose to minimize service latency and migration cost. They formulate the problem as a stochastic optimization problem and use Lyapunov framework to solve it. In their model, historical measurements of undesirous migration costs were used to construct a virtual queue with initial queue backlog of 0. Any queue backlog at time slot t for the virtual queue is taken as the exceeded cost of service migration. The stability of the virtual queue is used as a control to keep the migration cost minimized.

The Lyapunov optimization framework can been used to convert an optimization problem to a queue-based problem [63]. It has proven beneficial in service migration placement problems [63], [86]. The Lyapunov optimization framework works on the basis of a queuing system, which does not guarantee a solution that tends towards a global optimal one.

5) DIRECTED GRAPH NETWORK

The container placement problem and the topology of edge networks influence the use of directed graph network [16], [61]. A directed graph is used to convert the container placement problem to a flow-based problem. Figure 5 depicts this flow. End users U1 to U4 submit computing tasks, which are mapped to containers C1 to C4. The containers C1 to C4 are then mapped to edge nodes N1 to N4 as shown. The minimum number of paths is four representing the: i) the task request, ii) the task placement on a container, iii) the container placement on an edge node and iv), the completed task or response. For a migration problem, the number of paths may be more, depending on the number

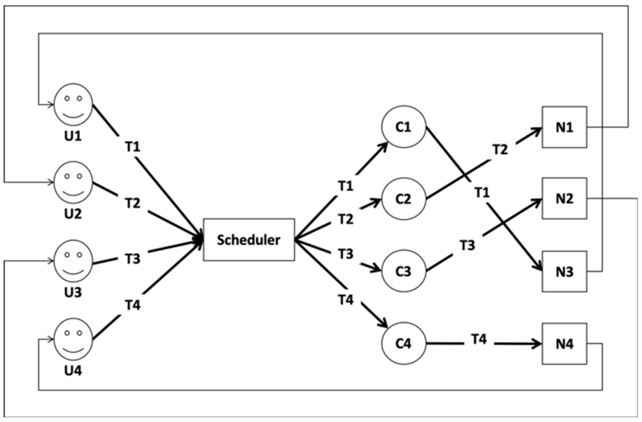

FIGURE 5. Mapping service of request to containers and mapping containers to edge nodes, using directed graph network.

of migrations used in servicing a request. This leads to a multi-level decision making process for the scheduler.

By modelling the container placement and migration problem as a directed graph, Zhou *et al.* [13] calculate the optimal flow path that minimizes latency, while authors Hu *et al.* [61] establish an optimal path that minimizes cost. One of the motivations for modelling the container placement problem with a directed graph is to convert the problem to an optimization flow problem such that an optimization flow-based algorithm can be used to solve it [61]. Examples of such algorithms are the cycle canceling algorithm and successive shortest path algorithm.

Undirected bipartite graph networks are non-flow based. Figure 6 can be used to describe this concept for mapping tasks (service requests) to containers. The set of tasks and containers make up the vertex pair. Tasks T1 to T5 are to be mapped to containers C1 to C5. The topology (Figure 6a) shows the possible mappings, depicted using the edges, i.e. the links. Task T1 can be mapped to containers C1 and C2; Task T2 can only be mapped to container C1 and so on. However, only one task can be mapped to a container, since a container, by definition, cannot service more than one task. A graph matching attempts to generate a mapping such that no two edges share the same vertex. The vertices (T2 and C3) without edges causes the mapping to be imperfectly matched. A perfect matching is where all tasks are mapped. A maximum matching attempts to map the largest number of tasks.

Graph edges may represent inter-edge node distance and bandwidth [36] and other network-specific parameters. In directed graph networks weights are assigned to edges. The goal is to optimize the total weights of all edges in a perfectly matched graph. There are graph-based algorithms, such as the Kuhn–Munkres algorithm, which can be used for solving the optimization problem.

In the development of their container scheduling model, authors in [11] have converted the task scheduling problem to a problem of weighted bipartite graph matching. They describe the use of bipartite graph matching. A bipartite

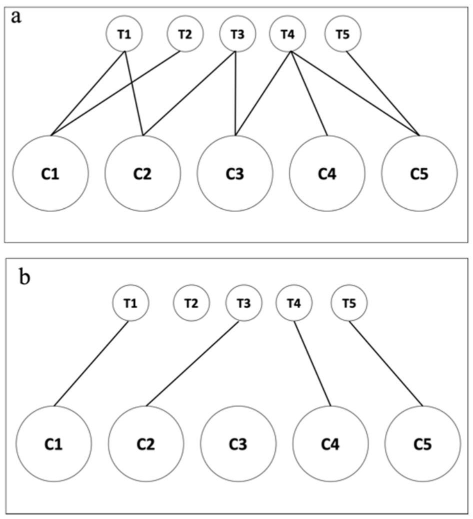

FIGURE 6. Mapping of service request to containers, using undirected graph network.

graph is such that the number of tasks must be equal to the number of containers. In a situation where there are more containers than tasks, the edge orchestrator removes the extra containers [11]. If the number of tasks is more than that of containers, hypothetical containers are added to make the number equal. Then a graph-based algorithm can be applied to find the optimal solution of the problem of weighted bipartite graph matching. The edges for the hypothetical containers are assigned a weights of zero so that they are not executed by the algorithm, and thus will not be selected for scheduling [11].

A graph-based model is useful for mapping the edge network topology. A graph-based approach can also perform the concurrent container scheduling [61]. In an undirected bipartite graph, the matching increases in complexity as the number of vertices (tasks and containers) increase.

B. ALGORITHMS

Algorithms are very efficient in solving many complex problems. The scheduling frameworks described so far have been used to convert the container placement problem to a problem that can be solved using an algorithm. Some of the formulated problems are combinatorial optimization problems (see Section IV-A-1). They have also been proven to be NP-hard [13], [85], [86]. Such problems can be solved using approximation algorithms or heuristics algorithms [100]. These algorithms are able to solve complex optimization problems very quickly and speed up the decision-making process of the scheduler. If the algorithm takes too long to arrive at a solution, the scheduler will be tardy in the container placement decisions, which can increase the overall latency of computation and reduce QoS. Algorithms, like frameworks, play a pivotal role in container scheduling models.

Heuristic refers to an approach that seeks an optimal or near-optimal solution as quickly as possible, but does not guarantee that it is the most optimal. Heuristic algorithms

Algorithm 1 Example Pseudocode for Heuristic Algorithm

```
Input = set of inputs, computation to run
Output = optimal solution at each iteration
1- Begin with initial or most recent optimal solution
2- Run computation and generate new solution
3- If new solution is better than most recent optimal solution, choose new solution, otherwise retain most recent solution
4- End
```

assist to balance the trade-off between time complexity of computation and accuracy.

A heuristic algorithm searches for an optimal solution at every iteration. If it finds a better solution, it makes the current solution as the optimal one, otherwise, the algorithm keeps the most recently found solution [101]. The pseudocode for a heuristic algorithm is described in Algorithm 1.

Fahs and Pierre, [80], attempt to limit the number of application replicas by scheduling where they are placed. They use a heuristic algorithm to schedule each replica placement. One of the objectives of the scheduler was to place replicas closer to the end user, using a proximity metric. Another objective was to minimize the load imbalance in the cluster arising from replica placement or migration. In the heuristic, application replicas are ordered according to usage. The frequently used replicas are more favored. The heuristic also lists replicas according to their proximity measure. Replicas and nodes closer to the main sources of traffic are more favored. Their algorithm iterates through the ordered lists and attempts to find a solution that optimizes the objective function.

The iteration stops when an improvement is found for the solution by a pre-defined value. If no improvement is found, then the current placement is maintained.

Asensio *et al.* [16] present a heuristic algorithm in their design of a concurrent container scheduling model. In their model, high scores are given to applications that require a specific runtime environment, applications that require a low number of pods and applications that do not have to be isolated. For edge nodes, high scores are given to nodes having large resources. The heuristic algorithm is used to define the scoring functions to rank container applications and nodes. In each iteration of the algorithm run time, a percentage of the highest ranked applications is sorted in a random manner, and the algorithm tries to allocate all these applications to nodes in an ordered manner.

The Greedy search algorithm is a well-known approximation algorithm, based on the heuristic concept. The Kubernetes scheduler uses a greedy algorithm. Algorithm 2 is a pseudocode that describes the greedy algorithm, for scheduling the placement of containerized data blocks on edge nodes. In the pseudocode, a set of containerized data blocks *n,* each with size s are to be placed in an edge node with space E. The objective is to maximize the value of all data blocks that can be placed in the edge node, where greater value is placed on larger data blocks.

Algorithm 2 Example Pseudocode for Greedy Algorithm

```
Input = set $A$ of $n$ data blocks each with data size $s$
Output = maximize $n$ in space $E$
1- Begin
2- Rank the data blocks in order of size from largest to least $s_{max} \leq s'_{max} \leq s''_{max} \leq s'''_{max} \cdots$
3- Choose a data block $n_{max}$ that has the largest data size $s_{max}$
4- Add $n_{max}$ to $E$
5- Remove $n_{max}$ from $A$ to obtain new $A$
6- While $E$ is not maximized, repeat steps 2- 5 for every $s'_{max}, s''_{max}, s'''_{max} \ldots$ until $E$ is maximized
7- End
```

The first fit method is a very common scheduling approach in container placement decisions. In first fit approach, the scheduler attempts to place a container on the first edge node that can accommodate it [85]. This is a greedy concept and is used by the Docker Swarm scheduler.

Mendes [83] describe the use of a greedy algorithm for their energy-aware container placement scheduling model. Their algorithm uses the energy efficiency levels at the nodes to schedule container placement. In their scheduling model they define three levels of resource utilization: low, desired and degradation energy efficiency. Their model utilizes a host registry that lists, in descending order, the total resource utilization of each host in each level. Priority placement is given to the first elements (nodes) in the list of low energy efficiency nodes so that those nodes can move to the desired level of energy efficiency

The min-min algorithm is a heuristic-based algorithm that mimics the greedy approach. In min-min, the minimum completion time for each task is computed with respect to all available edge nodes [102]. The task with the overall minimum completion time is selected and assigned to the corresponding edge node. The newly mapped task is removed from the list, and the process iterates until all tasks are mapped [102]. The main limitation with the min-min algorithm is that it can lead to load imbalance because it attempts to schedule small tasks first [102]. Chen *et al.* [103] use a min-min algorithm in their container scheduling model. In their model, containers with the least increase in energy consumption are prioritized.

Tabu-search algorithm is a heuristic-based algorithm. It searches through a set of solutions (from an updated tabu list) rather than one solution at each iteration, i.e. it employs a neighborhood search. It uses memory to store previous solutions arrived at, so these solutions can be retrieved at future iterations. The authors in [11] used Tabu-search to solve their knapsack problem formulation for the optimal placement of containerized data blocks in edge computing.

Another heuristic-based algorithm is Ant Colony. It can be used for finding the optimal paths through directed graph networks [39]. The Ant Colony algorithm is non-iterative, rather it uses random sampling probability to take decisions. In a study done by [39], the authors utilize Ant Colony algorithm in their container migration mechanism for balancing workload in edge servers. The problem is formulated as a multi-objective optimization problem. Ant Colony algorithm is used to solve the optimization problem while updating the global pheromone of all migrations.

Particle swarm optimization, is yet another heuristic-based algorithm. It has been used by Fan *et al.* [104], where they have formulated the container placement problem as a multi-objective optimization problem, with the joint objectives to minimize latency, minimize the number of failed requests, and minimize the load imbalance within the cluster. They describe the use of particle swarm optimization in solving the formulated optimization problem.

Like particle swarm and ant colony, the bacteria foraging algorithm is a population-based, nature-inspired metaheuristic algorithm. Sobhanayak *et al.* [66] proposed a scheduling model that aims to maximize resource utilization in the edge network. They formulate the optimal placement of tasks as an optimization problem and use a Hybrid Bacteria Foraging algorithm to solve it. The main benefit of using the algorithm is that it gives a feasible subjective solution in polynomial time.

The Kuhn-Munkres Hungarian algorithm is a graph-based algorithm, which can be used to find maximum-weight matchings in bipartite graphs, see applications in [11] and [86]. The Kuhn-Munkres algorithm adds time complexity of O(|V|3), where V is the number of vertices.

A directed graph network is also a flow-based network. Hu *et al.* [61] model a concurrent container scheduling problem as a minimum cost flow problem. Minimum cost flow problems are solved using the cycle cancelling algorithm [105]. Hu *et al.* [61] aim to minimize the total cost of the flow in the directed graph network. They used the cycle cancelling algorithm to find an optimal solution.

Reinforcement learning is a classification-based algorithm concerned with how intelligent agents (such as a container scheduler) should take actions in an environment, in order to maximize the notion of cumulative reward. Reinforcement learning can be used to solve complex intractable problems that have been modelled using MDP. The purpose of reinforcement learning is for the agent to learn an optimal, or near-optimal policy that maximizes the user-specified reward function. In [84], authors model a microservice coordination problem using an MDP. Reinforcement learning, with a greedy selection policy, was used to find the optimal decision. Authors in [85] propose to maximize system utility in the joint scheduling of delay-sensitive and computation-oriented tasks. They model the problem as an optimization problem and use Deep reinforcement learning with deep deterministic policy gradient to arrive at an optimal solution. More recently, Han *et al.* [81] propose a scheduling system that decentralizes the scheduling decisions to the multiple master nodes. To enable this, they use a multi-agent deep reinforcement learning algorithm.

TABLE 2. Container placement scheduling models in edge computing: A matching of algorithms to frameworks for.

Framework	Algorithm
Optimization modelling	Heuristic (Maheshwari et al. 2018) Metaheuristic (Ma, Shao et. al., 2020)
Multi-dimensional knapsack problem	Heuristic (Araldo et al., 2020; Faticanti et a., 2029; Tang et al., 2020) Metaheuristic (Li, Bai et al., 2019)
Markov decision process	Reinforcement learning (Wang, Guo et al., 2019; Tang, Zhou et al., 2018; Han et al., 2021)
Lyapunov function	Heuristic (Zhao, Wang et al., 2020) Hungarian (Ge et al., 2020)
Directed Graph Network	Metaheuristic (Zhou et al., 2020) Cycle cancelling (Hu et al.,2020) Heuristic (Asensio et al., 2020)
Undirected Graph Network	Metaheuristic (Li, Bai et al., 2019) Heuristic (Rausch et al., 2021)

VI. DISCUSSION

Much research effort has been expended on the development of scheduling models for container placement and migration in edge computing. This study accentuates two critical aspects of the model building process, namely the frameworks for modelling the scheduling problem and the types of algorithms that are used. The frameworks convert the container placement scheduling problem to one that can be solved using an algorithm. Approaches that have been used to frame the container placement scheduling problem include: Optimization Modelling, Multi-dimensional Knapsack Problem, Markov Decision Process, Lyapunov Optimization Modelling and Graph Network. The container placement scheduling problem is a complex NP-hard problem and warrants the use of such modelling frameworks and subsequent algorithms. The types of algorithms that have been used include: heuristic (and metaheuristics) graph-based and reinforcement learning.

One would expect to find definitive patterns emerging, i.e. patterns that link frameworks with algorithms. Table 2 has been used to register the few studies where patterns were observed, but still some discrepancies exist. For instance, reinforcement learning algorithm has typically been used where the container placement problem was modelled using Markov Decision Process.

However, Zhang *et al.* [85] model a delay-sensitive and computation-oriented container placement problem using an optimization modelling framework. They use a reinforcement learning-based algorithm to solve the formulated problem. Although the types of algorithms are mainly heuristic, graph-based and reinforcement learning, the variety of algorithms that have been used is unexplainable. On the other hand, findings indicate that the container placement problem is majorly a joint optimization problem. Most of the scheduling models are based on optimization models. All of the frameworks necessitate solving an optimization problem.

Most of the edge container placement scheduling decisions are taken on the basis that each request is assigned to a single container and each container is assigned to no more than one edge node. Computations such as distributed or parallel computing sometimes require multi-container, multi-node placement. There is paucity of scheduling models that consider this type of container placement problem [15].

Container migration is a unique type of container placement, because the placement is to relocate an already running container from one edge node to another. This type of container placement is rarely covered within the scheduling models so far advanced [13]. It is unclear how migrating containers are prioritized and placed ahead of new requests.

Containerized edge computing is predicted to grow exponentially. As tasks, containers and edge devices increase, the edge container placement scheduler will be more challenged in the near future. Currently, most of the scheduling models rely on a centralized decision-making system [15]. These centralized servers will become overloaded as edge computing continues in its growth trajectory. This will necessitate more clusters, further increasing the workload for the orchestrator in managing many master nodes. This can lead to increased latency because in a centralized scheduling system, additional time is taken to upload system states and wait for dispatch decisions [81]. Decentralizing the scheduling decisions to multiple edge servers is inevitable [98]. The work by Han *et al.* [81] is among the very few efforts in this area. They describe, in detail, a decentralized Kubernetes-oriented container scheduling model for edge servers. In their system, they deploy a scheduling agent on multiple master nodes. Experiments show that the scheduling delay of centralized service orchestration is almost nine times that of the decentralized approach. Casquero *et al.* [78] describe a custom scheduler for Kubernetes orchestrator that distributes the decision logic of the scheduler among edge nodes. Their *agentified* scheduler is supposed to reduce the workload at the control plane of the Kubernetes server. The node filtering and node ranking function normally done by the server is undertaken by agents embedded in the edge nodes. The authors built a multi-agent scheduling platform which receives the node filtering information from all nodes. Then node ranking is fulfilled through negotiation among the agents in the filtered edge nodes. Experimental tests showed that the scheduling times for the agentified scheduler were lower than those from a centralized scheduler. Wang *et al.* [98] investigate the dynamic service migration in mobile edge computing. They model the problem using Markov Decision Process and develop an algorithm for computing the optimal service placement policy. Their algorithm is based on a modified policy iteration approach. Although they investigate a non-containerized edge server, their approach can be extended to the service placement of container-based applications. While they focused on a centralized control mechanism, their model is applicable in a decentralized system as well. The works in along the research stream relating to decentralized schedulers are not yet definitive. More work is needed in this area.

The container placement problem is mostly formulated on the basis that the edge nodes are stationary. Mobile edge computing is likely to grow as autonomous vehicles and augmented reality applications become more widespread. Stationary edge nodes are mostly sparsely distributed, and this affects mobile edge computing, such that edge services have to be partially fulfilled on the cloud in some instances [10], [67]. Stationary edge nodes will be challenged in the near future, which makes a case for mobile edge nodes, such as using vehicular edge computing [86]. Vehicular edge computing allows edge computing to be ubiquitous, but scheduling the placement and migration of containers in moving edge nodes opens up new challenges and novel research prospects.

VII. CONCLUSION

In this study, the container placement scheduling problem in edge computing has been investigated. First the concept of container placement and migration is clarified. Then, frameworks and algorithms that have been used to build the scheduling models were reviewed. A number of key findings were reported. The container placement problem in mostly abstracted using multi-objective optimization models or graph network models, to convert the problem to one that can be solved using an algorithm. The scheduling algorithms are predominantly heuristic-based algorithms, which are able to arrive at sub-optimal solutions very quickly. There is paucity of container scheduling models that consider distributed edge computing tasks. Research in decentralized scheduling systems is gaining momentum. The future outlook is in scheduling containers for mobile edge nodes.

This study has focused on the frameworks and algorithms that have been used in building container scheduling models for edge servers. The study showed that optimization modelling frameworks and heuristic-based algorithms are ubiquitous. This may suggest that optimization modelling and heuristic algorithms are more advantageous than others. The current study is theoretical in nature, and would be limited to only a theoretically-based comparative analysis. An empirical study is needed to fulfil a more factual comparative analysis.

ACKNOWLEDGMENT

The author is deeply grateful to the three anonymous reviewers, whose comments and suggestions contributed to the enhancement of the final version of the article.

REFERENCES

[1] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan, "Survey of techniques and architectures for designing energy-efficient data centers," *IEEE Syst. J.*, vol. 10, no. 2, pp. 507–519, Jun. 2016.

[2] K. Nakanishi, F. Suzuki, S. Ohzahata, R. Yamamoto, and T. Kato, "A container-based content delivery method for edge cloud over wide area network," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Jan. 2020, pp. 568–573.

[3] M. Salehe, Z. Hu, S. H. Mortazavi, I. Mohomed, and T. Capes, "VideoPipe: Building video stream processing pipelines at the edge," in *Proc. 20th Int. Middleware Conf. Ind. Track*, Dec. 2019, pp. 43–49.

[4] R. Meulen, "What edge computing means for infrastructure and operations leaders," Web Post Infrastruct. Oper., Garner, Tech. Rep., 2018. Accessed: Mar. 20, 2021. [Online]. Available: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders

[5] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for Internet of Things realization," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2961–2991, Jun. 2018.

[6] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A survey on service migration in mobile edge computing," *IEEE Access*, vol. 6, pp. 23511–23528, 2018.

[7] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85714–85728, 2020.

[8] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on raspberry pi clusters," in *Proc. IEEE 4th Int. Conf. Future Internet Things Cloud Workshops (FiCloudW)*, Aug. 2016, pp. 117–124.

[9] P. Kayal, "Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope," in *Proc. IEEE 6th World Forum Internet Things (WF-IoT)*, Jun. 2020, pp. 1–6.

[10] A. Araldo, A. D. Stefano, and A. D. Stefano, "Resource allocation for edge computing with multiple tenant configurations," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 1190–1199.

[11] C. Li, J. Bai, and J. Tang, "Joint optimization of data placement and scheduling for improving user experience in edge computing," *J. Parallel Distrib. Comput.*, vol. 125, pp. 93–105, Mar. 2019.

[12] P. Zhao, P. Wang, X. Yang, and J. Lin, "Towards cost-efficient edge intelligent computing with elastic deployment of container-based microservices," *IEEE Access*, vol. 8, pp. 102947–102957, 2020.

[13] A. Zhou, S. Wang, S. Wan, and L. Qi, "LMM: Latency-aware micro-service mashup in mobile edge computing environment," *Neural Comput. Appl.*, vol. 32, no. 19, pp. 15411–15425, Oct. 2020.

[14] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4228–4237, May 2020.

[15] V. Cardellini, F. L. Presti, M. Nardelli, and F. Rossi, "Self-adaptive container deployment in the fog: A survey," in *Proc. Int. Symp. Algorithmic Aspects Cloud Comput.* Cham, Switzerland: Springer, 2019, pp. 77–102.

[16] A. Asensio, X. Masip-Bruin, J. Garcia, and S. Sánchez, "On the optimality of concurrent container clusters scheduling over heterogeneous smart environments," *Future Gener. Comput. Syst.*, vol. 118, pp. 157–169, May 2021.

[17] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, "Edge cloud offloading algorithms: Issues, methods, and perspectives," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 1–23, Feb. 2019.

[18] S. Chen, Q. Li, M. Zhou, and A. Abusorrah, "Recent advances in collaborative scheduling of computing tasks in an edge computing paradigm," *Sensors*, vol. 21, no. 3, p. 779, Jan. 2021.

[19] M. S. U. Islam, A. Kumar, and Y.-C. Hu, "Context-aware scheduling in fog computing: A survey, taxonomy, challenges and future directions," *J. Netw. Comput. Appl.*, vol. 180, Apr. 2021, Art. no. 103008.

[20] K. Matrouk and K. Alatoun, "Scheduling algorithms in fog computing: A survey," *Int. J. Netw. Distrib. Comput.*, vol. 9, no. 1, pp. 59–74, 2021.

[21] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 17, p. e5668, Sep. 2020.

[22] P.-J. Maenhaut, B. Volckaert, V. Ongenae, and F. De Turck, "Resource management in a containerized cloud: Status and challenges," *J. Netw. Syst. Manage.*, vol. 28, no. 2, pp. 197–246, Apr. 2020.

[23] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *J. King Saud Univ.-Comput. Inf. Sci.*, Mar. 2021, doi: 10.1016/j.jksuci.2021.03.002.

[24] Z. Tao, Q. Xia, Z. Hao, C. Li, L. Ma, S. Yi, and Q. Li, "A survey of virtual machine management in edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1482–1499, Aug. 2019.

[25] R. Du, K. Xu, and X. Liang, "Multiattribute evaluation model based on the KSP algorithm for edge computing," *IEEE Access*, vol. 8, pp. 146932–146943, 2020.

[26] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in intermittently-powered IoT devices," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 3, pp. 1–23, Jul. 2017.

[27] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran, and M. Shoaib, "Bringing computation closer toward the user network: Is edge computing the solution?" *IEEE Commun. Mag.*, vol. 55, no. 11, pp. 138–144, Nov. 2017.

[28] H. M. Fard, R. Prodan, and F. Wolf, "A container-driven approach for resource provisioning in edge-fog cloud," in *Proc. Int. Symp. Algorithmic Aspects Cloud Comput.* Cham, Switzerland: Springer, 2019, pp. 59–76.

[29] V. Divya and R. S. Leena, "Docker based intelligent fall detection using edge-fog cloud infrastructure," *IEEE Internet Things J.*, early access, Dec. 4, 2020, doi: 10.1109/JIOT.2020.3042502.

[30] A. Stanciu, "Blockchain based distributed control system for edge computing," in *Proc. 21st Int. Conf. Control Syst. Comput. Sci. (CSCS)*, May 2017, pp. 667–671.

[31] P.-H. Tsai, H.-J. Hong, A.-C. Cheng, and C.-H. Hsu, "Distributed analytics in fog computing platforms using tensorflow and kubernetes," in *Proc. 19th Asia–Pacific Netw. Oper. Manage. Symp. (APNOMS)*, Sep. 2017, pp. 145–150.

[32] N. Tellez, M. Jimeno, A. Salazar, and E. D. Nino-Ruiz, "Container-based architecture for optimal face-recognition tasks in edge computing," in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, Nov. 2019, pp. 301–303.

[33] J. Tang, R. Yu, S. Liu, and J.-L. Gaudiot, "A container based edge offloading framework for autonomous driving," *IEEE Access*, vol. 8, pp. 33713–33726, 2020.

[34] A. T. Z. Kasgari and W. Saad, "Stochastic optimization and control framework for 5G network slicing with effective isolation," in *Proc. 52nd Annu. Conf. Inf. Sci. Syst. (CISS)*, Mar. 2018, pp. 1–6.

[35] S. R. Chaudhry, A. Palade, A. Kazmi, and S. Clarke, "Improved QoS at the edge using serverless computing to deploy virtual network functions," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 10673–10683, Oct. 2020.

[36] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Gener. Comput. Syst.*, vol. 114, pp. 259–271, Jan. 2021.

[37] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *Proc. IEEE 17th Int. Symp. A World Wireless, Mobile Multimedia Netw. (WoWMoM)*, Jun. 2016, pp. 1–6.

[38] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. 2014. Accessed: Jan. 14, 2021. [Online]. Available: https://dockerbook.com/TheDockerBook_sample.pdf

[39] Z. Ma, S. Shao, S. Guo, Z. Wang, F. Qi, and A. Xiong, "Container migration mechanism for load balancing in edge network under power Internet of Things," *IEEE Access*, vol. 8, pp. 118405–118416, 2020.

[40] A. Krylovskiy, "Internet of Things gateways meet Linux containers: Performance evaluation and discussion," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Dec. 2015, pp. 222–227.

[41] P. Mendki, "Docker container based analytics at IoT edge video analytics usecase," in *Proc. 3rd Int. Conf. Internet Things, Smart Innov. Usages (IoT-SIU)*, Feb. 2018, pp. 1–4.

[42] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Proc. Int. Conf. Adv. Comput. Eng. Appl.*, Mar. 2015, pp. 342–346.

[43] B. I. Ismail, E. M. Goortani, M. B. A. Karim, W. M. Tat, S. Setapa, J. Y. Luke, and O. H. Hoe, "Evaluation of docker as edge computing platform," in *Proc. IEEE Conf. Open Syst. (ICOS)*, Aug. 2015, pp. 130–135.

[44] H. Manninen, V. Jääskeläinen, and J. O. Blech, "Performance evaluation of containerization platforms for control and monitoring devices," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2020, pp. 1061–1064.

[45] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *Proc. 3rd USENIX Workshop Hot Topics Edge Comput. (HotEdge)*, 2020, pp. 1–7.

[46] Docker. *What is a Container*. Accessed: Feb. 18, 2021. [Online]. Available: https://www.docker.com/resources/what-container

[47] S. McCarty, "Choosing the right container base image for your application," Tech. Rep., 2019. Accessed: Feb. 23, 2021. [Online]. Available: https://crunchtools.com/files/2019/05/Choosing-the-right-container-base-image-for-your-application.pdf

[48] Q. Qu, R. Xu, S. Y. Nikouei, and Y. Chen, "An experimental study on microservices based edge computing platforms," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 836–841.

[49] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proc. Int. Conf. Internet Things Design Implement.*, Apr. 2019, pp. 225–236.

[50] J. Chabas, G. Chandra, G. Sanchi, and M. Mitra. *New Demand, New Markets-What Edge Computing Means for Hardware Companies*. Accessed: Feb. 18, 2021. [Online]. Available: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/new-demand-new-markets-what-edge-computing-means-for-hardware-companies

[51] P. Kochovski and V. Stankovski, "Supporting smart construction with dependable edge computing infrastructures and applications," *Autom. Construct.*, vol. 85, pp. 182–192, Jan. 2018.

[52] V. P. Betancourt, B. Liu, and J. Becker, "Model-based development of a dynamic container-based edge computing system," in *Proc. IEEE Int. Symp. Syst. Eng. (ISSE)*, Oct. 2020, pp. 1–5.

[53] M. Al-Rakhami, A. Gumaei, M. Alsahli, M. M. Hassan, A. Alamri, A. Guerrieri, and G. Fortino, "A lightweight and cost effective edge intelligence architecture based on containerization technology," *World Wide Web*, pp. 1–20, May 2019, doi: 10.1007/s11280-019-00692-y.

[54] T. Tasci, J. Melcher, and A. Verl, "A container-based architecture for real-time control applications," in *Proc. IEEE Int. Conf. Eng., Technol. Innov. (ICE/ITMC)*, Jun. 2018, pp. 1–9.

[55] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner, "Container-based architecture for flexible industrial control applications," *J. Syst. Archit.*, vol. 84, pp. 28–36, Mar. 2018.

[56] Y. Song, J. Xie, Q. Huang, M. Wang, and J. Yu, "Design and implementation of turtle breeding system based on embedded container cloud," in *Proc. 2nd IEEE Adv. Inf. Manage., Commun., Electron. Automat. Control Conf. (IMCEC)*, May 2018, pp. 2531–2534.

[57] R. Senington, B. Pataki, and X. V. Wang, "Using docker for factory system software management: Experience report," *Procedia CIRP*, vol. 72, pp. 659–664, Jan. 2018.

[58] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. C. Otero, "A modular CPS architecture design based on ROS and docker," *Int. J. Interact. Des. Manuf.*, vol. 11, no. 4, pp. 949–955, Nov. 2017.

[59] Z. Y. Thean, V. V. Yap, and P. C. Teh, "Container-based MQTT broker cluster for edge computing," in *Proc. 4th Int. Conf. Workshops Recent Adv. Innov. Eng. (ICRAIE)*, Nov. 2019, pp. 1–6.

[60] J. Reijonen, M. Opsenica, R. Morabito, M. Komu, and M. Elmusrati, "Regression training using model parallelism in a distributed cloud," in *Proc. IEEE Int. Conf. Dependable, Autonomic Secure Comput., Int. Conf. Pervas. Intell. Comput., Int. Conf. Cloud Big Data Comput., Int. Conf. Cyber Sci. Technol. Congr. (DASC/PiCom/CBDCom/CyberSciTech)*, Aug. 2019, pp. 741–747.

[61] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao, "Concurrent container scheduling on heterogeneous clusters with multi-resource constraints," *Future Gener. Comput. Syst.*, vol. 102, pp. 562–573, Jan. 2020.

[62] J. Luo, L. Yin, J. Hu, C. Wang, X. Liu, X. Fan, and H. Luo, "Container-based fog computing architecture and energy-balancing scheduling algorithm for energy IoT," *Future Gener. Comput. Syst.*, vol. 97, pp. 50–60, Aug. 2019.

[63] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leung, "Dynamic service migration and workload scheduling in edge-clouds," *Perform. Eval.*, vol. 91, pp. 205–228, Sep. 2015.

[64] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.

[65] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 181, May 2021, Art. no. 103005.

[66] S. Sobhanayak, K. Jaiswal, A. K. Turuk, B. Sahoo, B. K. Mohanta, and D. Jena, "Container-based task scheduling for edge computing in IoT-cloud environment using improved HBF optimisation algorithm," *Int. J. Embedded Syst.*, vol. 13, no. 1, pp. 85–100, 2020.

[67] S. Taherizadeh, V. Stankovski, and M. Grobelnik, "A capillary computing architecture for dynamic Internet of Things: Orchestration of microservices from edge devices to fog and cloud providers," *Sensors*, vol. 18, no. 9, p. 2938, Sep. 2018.

[68] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, Oct. 2017, pp. 1–13.

[69] I. Farris, T. Taleb, A. Iera, and H. Flinck, "Lightweight service replication for ultra-short latency applications in mobile edge networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–6.

[70] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," *Sensors*, vol. 19, no. 7, p. 1488, Mar. 2019.

[71] S. Maheshwari, S. Choudhury, I. Seskar, and D. Raychaudhuri, "Traffic-aware dynamic container migration for real-time support in mobile edge clouds," in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS)*, Dec. 2018, pp. 1–6.

[72] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proc. 3rd USENIX Workshop Hot Topics Edge Comput. (HotEdge)*, 2020, pp. 1–7.

[73] *Kubernetes*. Accessed: Mar. 20, 2021. [Online]. Available: https://kubernetes.io

[74] A. A. Majeed, P. Kilpatrick, I. Spence, and B. Varghese, "Performance estimation of container-based cloud-to-fog offloading," 2019, *arXiv:1909.04945*. [Online]. Available: http://arxiv.org/abs/1909.04945

[75] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration," in *Proc. Linux Symp.*, vol. 2, 2008, pp. 85–90.

[76] (2021). *CRIU*. Accessed: Feb. 12, 2021. [Online]. Available: https://criu.org/Main_Page

[77] M. C. Ogbuachi, A. Reale, P. Suskovics, and B. Kovács, "Context-aware kubernetes scheduler for edge-native applications on 5G," *J. Commun. Softw. Syst.*, vol. 16, no. 1, pp. 85–94, Mar. 2020.

[78] O. Casquero, A. Armentia, I. Sarachaga, F. Perez, D. Orive, and M. Marcos, "Distributed scheduling in kubernetes based on MAS for fog-in-the-loop applications," in *Proc. 24th IEEE Int. Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2019, pp. 1213–1217.

[79] D. Haja, M. Szalay, B. Sonkoly, G. Pongracz, and L. Toka, "Sharpening kubernetes for the edge," in *Proc. ACM SIGCOMM Conf. Posters Demos*, 2019, pp. 136–137.

[80] A. J. Fahs and G. Pierre, "Tail-latency-aware fog application replica placement," in *Proc. Int. Conf. Service-Oriented Comput.* Cham, Switzerland: Springer, 2020, pp. 508–524.

[81] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. M. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," 2021, *arXiv:2101.06582*. [Online]. Available: http://arxiv.org/abs/2101.06582

[82] W. Wong, A. Zavodovski, P. Zhou, and J. Kangasharju, "Container deployment strategy for edge networking," in *Proc. 4th Workshop Middleware Edge Clouds Cloudlets*, 2019, pp. 1–6.

[83] S. Mendes, J. Simão, and L. Veiga, "Oversubscribing micro-clouds with energy-aware containers scheduling," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2019, pp. 130–137.

[84] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Trans. Mobile Comput.*, vol. 20, no. 3, pp. 939–951, Mar. 2021.

[85] F. Zhang, Z. Tang, J. Lou, and W. Jia, "Online joint scheduling of delay-sensitive and computation-oriented tasks in edge computing," in *Proc. 15th Int. Conf. Mobile Ad-Hoc Sensor Netw. (MSN)*, Dec. 2019, pp. 303–308.

[86] S. Ge, M. Cheng, X. He, and X. Zhou, "A two-stage service migration algorithm in parked vehicle edge computing for Internet of Things," *Sensors*, vol. 20, no. 10, p. 2786, May 2020.

[87] A. Singh, G. S. Aujla, and R. S. Bali, "Container-based load balancing for energy efficiency in software-defined edge computing environment," *Sustain. Comput., Informat. Syst.*, vol. 30, Jun. 2021, Art. no. 100463.

[88] S. Wang, "Dynamic service placement in mobile micro-clouds," Ph.D. dissertation, Imperial College London, London, U.K., 2015.

[89] Y. Song, C. Zhang, and Y. Fang, "Multiple multidimensional knapsack problem and its applications in cognitive radio networks," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Nov. 2008, pp. 1–7.

[90] P. C. Chu and J. E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *J. Heuristics*, vol. 4, no. 1, pp. 63–86, Jun. 1998.

[91] I. Ketykó, L. Kecskés, C. Nemes, and L. Farkas, "Multi-user computation offloading as multiple knapsack problem for 5G mobile edge computing," in *Proc. Eur. Conf. Netw. Commun. (EuCNC)*, Jun. 2016, pp. 225–229.

[92] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Cutting throughput with the edge: App-aware placement in fog computing," in *Proc. 6th IEEE Int. Conf. Cyber Secur. Cloud Comput. (CSCloud)/5th IEEE Int. Conf. Edge Comput. Scalable Cloud (EdgeCom)*, Jun. 2019, pp. 196–203.

[93] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Hoboken, NJ, USA: Wiley, 2014.

[94] A. Ksentini, T. Taleb, and M. Chen, "A Markov decision process-based service migration procedure for follow me cloud," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2014, pp. 1350–1354.

[95] M. Tokic and G. Palm, "Value-difference based exploration: Adaptive control between epsilon-greedy and softmax," in *Proc. Annu. Conf. Artif. Intell.* Berlin, Germany: Springer, 2011, pp. 335–346.

[96] L. Tang, H. Yang, R. Ma, L. Hu, W. Wang, and Q. Chen, "Queue-aware dynamic placement of virtual network functions in 5G access network," *IEEE Access*, vol. 6, pp. 44291–44305, 2018.

[97] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Trans. Services Comput.*, vol. 12, no. 5, pp. 712–725, Sep. 2019.

[98] 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 Trans. Netw.*, vol. 27, no. 3, pp. 1272–1288, Jun. 2019.

[99] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2333–2345, Oct. 2018.

[100] E. Aarts, E. Aarts, and J. Lenstra, *Local Search in Combinatorial Optimization*. Princeton, NJ, USA: Princeton Univ. Press, 2003.

[101] H. Pirim, B. Eksioglu, and E. Bayraktar, *Tabu Search: A Comparative Study*. London, U.K.: INTECH Open Access Publisher, 2008.

[102] M.-Y. Wu, W. Shu, and H. Zhang, "Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems," in *Proc. 9th Heterogeneous Comput. Workshop (HCW)*, 2000, pp. 375–385.

[103] F. Chen, X. Zhou, and C. Shi, "The container scheduling method based on the min-min in edge computing," in *Proc. 4th Int. Conf. Big Data Comput.*, 2019, pp. 83–90.

[104] G. Fan, L. Chen, H. Yu, and W. Qi, "Multi-objective optimization of container-based microservice scheduling in edge computing," *Comput. Sci. Inf. Syst.*, vol. 18, no. 1, pp. 23–42, 2020.

[105] P. T. Sokkalingam, R. K. Ahuja, and J. B. Orlin, "New polynomial-time cycle-canceling algorithms for minimum-cost flows," *Networks*, vol. 36, no. 1, pp. 53–63, 2000.

OMOGBAI OLEGHE received the B.Sc. degree in mechanical engineering from the University of Lagos, Nigeria, in 1991, the M.B.A. degree from the Birmingham Business School, University of Birmingham, U.K., in 1996, and the M.Sc. degree in manufacturing consultancy and the Ph.D. degree in lean manufacturing and simulation modeling from Cranfield University, U.K., in 2013 and 2019, respectively. He has acquired a specialization in discrete event simulation, system dynamics modeling, machine learning, deep learning, and the Industrial Internet of Things infrastructures. He is a Visiting Lecturer at the Systems Engineering Department of the University of Lagos, Nigeria. He is also a Consultant on data-driven manufacturing. His current research focus is on the use of containers as the platform for the Industrial Internet of Things applications.