This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

AI-based Resource Provisioning of IoE Services in 6G: A Deep Reinforcement Learning Approach

Hani Sami*, Hadi Otrok†, Jamal Bentahar*, Azzam Mourad‡
*Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
†Center of Cyber-Physical Systems (C2PS), Department of EECS, Khalifa University, Abu Dhabi, UAE
‡Department of Computer science and Mathematics, Lebanese American University, Beirut, Lebanon
hani.sami@mail.concordia.ca, hadi.otrok@ku.ac.ae, bentahar@ciise.concordia.ca, azzam.mourad@lau.edu.lb

***Abstract*—Currently, researchers have motivated a vision of 6G for empowering the new generation of the Internet of Everything (IoE) services that are not supported by 5G. In the context of 6G, more computing resources are required, a problem that is dealt with by Mobile Edge Computing (MEC). However, due to the dynamic change of service demands from various locations, the limitation of available computing resources of MEC, and the increase in the number and complexity of IoE services, intelligent resource provisioning for multiple applications is vital. To address this challenging issue, we propose in this paper IScaler, a novel intelligent and proactive IoE resource scaling and service placement solution. IScaler is tailored for MEC and benefits from the new advancements in Deep Reinforcement Learning (DRL). Multiple requirements are considered in the design of IScaler's Markov Decision Process. These requirements include the prediction of the resource usage of scaled applications, the prediction of available resources by hosting servers, performing combined horizontal and vertical scaling, as well as making service placement decisions. The use of DRL to solve this problem raises several challenges that prevent the realization of IScaler's full potential, including exploration errors and long learning time. These challenges are tackled by proposing an architecture that embeds an Intelligent Scaling and Placement module (ISP). ISP utilizes IScaler and an optimizer based on heuristics as a boot-strapper and backup. Finally, we use the Google Cluster Usage Trace dataset to perform real-life simulations and illustrate the effectiveness of IScaler's multi-application autonomous resource provisioning.**

***Index Terms*—Resource Provisioning, Deep Reinforcement Learning (DRL), Service Placement, Resource Scaling, 5G, 6G, AI, Internet of Everything (IoE).**

I. Introduction

Context: The main applications supported by 5G include the augmented and virtual reality, smart vehicles, drones, millions of connected Internet of Things (IoT) devices for industrial, medical, and smart city applications [1]. Despite the promises that 5G offers, researchers are highlighting the need for 6G with a terabyte of data rate per second, which will expand 5G's capabilities and open the door for supporting a new generation of applications. These applications include services for supporting the Internet of Everything (IoE) and Artificial Intelligence's (AI) distributed learning [2]. In fact, authors in [3] propose an AI-enabled intelligent architecture for smart knowledge discovery in the 6G network, allowing the integration of AI algorithms in the wireless communication stack. The increase in IoT devices and users' requests for services leads to high volumes of data, which can be employed by AI agents to build knowledge towards solving challenging problems. These problems usually require human intervention, such as real-time wireless network and computing resource management for 6G environments.

The fog and edge nodes offer low communication latency for users and applications in the form of services. Mobile Edge Computing (MEC) is the scientific term for edge nodes that supports 5G's services [4]. While the addition of MEC is a point of consideration in 5G networks, MEC will be one of the building blocks of the future 6G architecture due to the increasing need for computation support for multiple applications [5]. Furthermore, because of the dynamic change in demands, the heterogeneous types of resources for MEC servers, and the need for fast service updates, the container-based hosting technology is proven to be suitable for orchestration and scaling in the MEC environments [6], [7].

Motivation: Aware of the increase in the number of heterogeneous IoE services, the limitation of MEC computing resources, and the dynamic change of service demands from different types of users, dynamic resource management of IoE services and MEC servers is essential in 5G and 6G [8], [9]. For instance, the increase of road traffic between residual and business areas requires scaling and orchestration of the autonomous driving services to accommodate for the dynamic changes of demands [10], [11]. Other examples of services include vehicular network management [12], [13] and unmanned aerial vehicles support for mobility [14], [15]. Performing manual scaling for such applications is not feasible in the long run. Therefore, there is a need for a joint intelligent resource scaling and service placement solution. Such a solution should be automated using AI, can be deployed in existing architectures, and produces decisions while considering multiple applications. Moreover, the intelligent solution is supposed to utilize containers due to their suitability for service updates. Furthermore, it is crucial to mention that containers of multi-application hosted in 5G or 6G environments are managed in a cluster-based architecture using an orchestration tool like Kubernetes [16]. Henceforth, it is sufficient to develop a resource management solution that works in container-based multi-application clusters to support the cellular networks' environment. Consequently, our main objective in this paper is to develop an effective and intelligent scaling and placement solution.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

Problems & Challenges: An effective auto-scaling and placement solution requires an intelligent model capable of predicting the users' demand or the usage of resources by various applications. Proactive provisioning, which prepares what the application needs as resources in the near future, is essential to avoid the time for initialing the Kubernetes cluster and updating IoE services. Resource scaling decisions should combine horizontal and vertical scaling for more optimized resource usage [17]. In short, horizontal and vertical scaling imply adding/removing service instances to/from running hosts, and adjusting the amount of resources used by a running instance respectively. Unfortunately, existing auto-scaling solutions do not have robust models for predicting the change in demands for services [16]. Moreover, hosts running the application instance offer a certain volume of resources that are subject to change depending on other running applications. Performing scaling on hosts with varying resource availability can cause resource overflow and application downtime. Furthermore, horizontal scaling creates new instances of services that should be placed on the correct host of a cluster, following a specific set of edge computing objectives [18]. Moreover, existing solutions are targeting a single application or service for scaling [19][18]. However, it is important to study multi-application scaling in the same cluster for achieving combined management of resources [20]. Finally, a large enterprise application may run a large cluster and many applications that require scaling, demanding a scalable solution.

To sum up, the limitations of existing auto-scaling solutions can be categorized as follows:

1) Unstable models for predicting the change of demand or resource usage by applications running on MEC servers.
2) A prediction model for the change of available resources on MEC servers is still not explored.
3) A service placement scheme for scaled services is not offered in the MEC context.
4) Multi-application scaling is not supported.
5) A scalable resource provisioning solution is not studied for large MEC clusters.

Aware of the above limitations and cooping with 1) the Deep Reinforcement Learning (DRL) advancements in the resource management field [21], and 2) the potential of having an AI fostered environment in 6G [22], exploring the use of DRL as a resource management solution is promising. There are some proposals in the literature that use Reinforcement Learning (RL) to construct an auto-scaler [18][19][23][24], however, they do not comply with 6G environment requirements. In [18], the authors proposed Dyna-Q, a solution that employs a model-based RL algorithm. The main problem with such an approach is the assumption that the transition probability matrix of the environment is given, which is not applicable in real life. Besides, the proposed MDP design in the literature does not consider the change in resources and the need for a service placement technique and does not scale for large inputs by causing memory issues. In the case of resource scaling, mistakes in producing decisions are not permitted due to the possibility of affecting the application availability. Therefore, the Quality of Service (QoS) and the Quality of Experience (QoE) are consequently affected. In summary, the challenges for building a DRL-based solution to perform auto-scaling:

1) A flexible and scalable MDP design that can meet the described auto-scaling requirements is required.
2) The probability transition matrix of the environment is unknown, hence a model-free RL algorithm must be built.
3) A model-free RL algorithm poses errors at the learning stages that should be avoided in scaling.

Contributions: To overcome these challenges and limitations, we propose in this paper IScaler, a DRL-based resource scaling and service placement solution combined with a suitable architecture for integration in clustering environments. IScaler is an extension of our previous work in [25], where horizontal scaling of a single application using the SARSA RL algorithm was proposed. The MDP design of IScaler is well studied to consider predicting the change in user demands reflected by the resource usage and the change of available resources on hosting nodes in the cluster. The efficient IScaler predictions allows performing proactive decisions. Moreover, the service placement solution is embedded in the state representation of the MDP and performs combined horizontal and vertical scaling in the action space. IScaler uses a custom-built model-free DRL algorithm that utilizes our designed MDP to build an optimal control policy. We also propose embedding IScaler in an Intelligent Scaling and Placement module (ISP) module that runs IScaler, an optimizer module (thereafter called Optimizer), and a solution switch module (thereafter called Solution Switch). The optimizer runs a heuristic-based solution to perform scaling when IScaler is not ready. Once IScaler learning converges, the Solution Switch is triggered to shift from the use of the Optimizer to start executing IScaler's decision in the environment. The contributions of this work are summarized as follows:

- A novel architecture that embeds ISP as a service for bootstrapping IScaler, our DRL-based solution.
- An MDP design for building IScaler, while respecting the MEC requirements.
- A custom DQN algorithm to build the novel IScaler.

A series of experiments using the Google Cluster Usage Trace dataset [26] are conducted. Through these experiments, we illustrate the ability of IScaler to perform optimal auto-scaling decisions in multi-application container-based clustering environments. We also experiment with the agent behavior during the changes in demand compared to the recent Dyna-Q solution [18]. Finally, we illustrate the advantages of using our ISP to overcome existing DRL limitations.

The remainder of this paper is divided as follows. In Section II, we present the existing academic and industry-related work. In Section III, we describe the problem statement of resource scaling. Our proposed approach embedding IScaler is demonstrated in Section IV. In Section V, we provide the novel MDP design of the scaling problem. Afterward, experiments are conducted for evaluating the performance of ISP and IScaler in Section VII. Finally, the conclusion wraps up the paper in Section VIII.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

II. Related Work

In this section, we overview the latest literature work that attempt to provide solutions related to dynamic resource provisioning in academia an industry. Table I summarizes and compares the features and limitations of each of the existing work compared to our proposed solution.

A. Classical Solutions

Classical solutions do not employ intelligent or machine-learning-based solutions. For instance, the authors in [30] deploys a space-search pruning algorithm to find the best edge server for migration and scaling. Despite that the complexity of the search algorithm can grow exponentially in the worst case, the solution has to wait for the demands to occur to make a decision. On the other side, the authors in [29] measure the system state and classify its workload into low, medium, and high based on predefined thresholds. Similar to [30], downtime or degradation of QoS can happen while scaling resources. In addition, the work in [31] proposes the use of a heuristics search algorithm to perform the resource scaling in a cloud environment. The limitation in [31] is the use of a heuristic-based solution to perform the scaling after the increase in demand occurs, which directly affects the QoS. Besides, a heuristic solution does not guarantee an optimal solutions.

B. Machine Learning Solutions

There are many recent proposals that utilize machine learning to solve wireless and resource management problems. In this section, we focus on the RL-based solutions, which outperform classical machine learning solutions due to their ability to perform linear and non-linear approximations for the state-action value function, adapt to environmental changes, and learn without prior knowledge. The main RL applications in the context of resource provisioning include network resource management, computational resource scaling, wireless network security, and content caching [32]. DRL solutions exist for each application under different fields, such as the internet of vehicles, unmanned aerial vehicles, cloud, edge computing, and cellular technologies (5G & 6G) [32]. With regard to network management, DRL is used for solving the problem of resource management for network slicing [33]. Besides, DRL is also exploited in [34] for securing the wireless communication at the physical layer by adjusting the agent's reflecting elements with a base station. In [35], RL-based linear function approximation is used for content caching on base stations in the context of 5G based on the change of users' demand. In the same context, the authors in [18] builds a Markov Decision Process (MDP) by defining the states, actions, reward function, and retrieving the probability transition function. In their work, a Model-Based RL solution is proposed to take the scaling decision while knowing the probability transition function through period updates. As shown in Table I, the main limitation of [18] is the risk of application downtime. Downtime can happen in two cases. First, the probability transition matrix extracted might not be representative enough of the dynamics of the environments. Therefore, wrong scaling decisions are possible to be made. Second, in case the state space grows, it becomes impossible to estimate the probability transition function because this will require huge memory on the processing machine.

More recently, several approaches, such as [27] and [28], are working on improving the prediction of workload forecasting, which leads to accurate scaling decisions if successful. However, these time-series forecasting approaches still look for seasonality and pattern in the data studied, which drains its accuracy in case new patterns are encountered.

C. Industry-Based Solutions

Dynamic resource scaling is mandatory in any clustering environment that hosts services or executes computing tasks. The leading industry companies that offer cloud platforms offer the service scaling feature. Examples of these cloud solutions are Google Cloud Platform, Microsoft Azure, and Amazon Web Services (AWS). These solutions integrate the Kubernetes clustering tool to benefit from the orchestration and embedded scaling features, thus offering new environments titled Google GKE [1], Azure AKS [2], and AWS EKS [3]. As shown in Table I, these environments offer vertical, horizontal, or both scaling, as well as availability because of the multi-zone hosting of services inside Kubernetes clusters. The main limitation of these environments is that the demands of services, such as resource load or response time experienced by the user, are not predicted. These solutions either rely on thresholds or manual configurations similar to Azure AutoScale, which runs per application instance [4]. Such problems are resolved by a platform-native solution for AWS titled Auto Scaling, which is independent of Kubernetes [5]. In AWS Auto Scaling, time series prediction takes place to be able to scale the application instance before actual demands occur. As mentioned earlier, this method is not reliable because the time-series model might not be able to capture seasonality or signature in the demands.

Resource load is not an accurate measure of the scaling decision in cluster-based environments; however, most of the recent research and industry approaches are utilizing this metric at the service instance and cluster levels. Moreover, as shown in Table I, some solutions do not consider service placement because either they perform vertical scaling only or they run the horizontal scaling inside the same hosting machine. Additionally, the resources available on the servers may vary because of hosting several other independent applications. However, resource prediction is not considered by the aforementioned state of the art solutions.

III. Problem Statement

Resource scaling is necessary for dynamic resource management of clustered computing environments. In the literature, resource scaling methods can either be vertical or horizontal for one micro-service. Horizontal scaling scales an instance

[1]https://cloud.google.com/kubernetes-engine
[2]https://azure.microsoft.com/en-us/services/kubernetes-service/
[3]https://aws.amazon.com/eks/
[4]https://azure.microsoft.com/en-ca/features/autoscale/
[5]https://aws.amazon.com/autoscaling/

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

TABLE I: Table of Comparison Between Latest Resource Scaling Solutions

Service	Horizontal Scaling	Vertical Scaling	Availability	Demands Prediction	Adaptation to Changes	Input Scaling	Service Placement	Resources Prediction	Model Bootstrapping
AWS EKS	-	✓	✓	-	-	✓	✓	-	-
AWS Auto Scaling	✓	-	✓	✓	-	-	-	-	-
Azure AKS	✓	-	✓	-	-	✓	✓	-	-
Azure AutoScale	✓	-	✓	-	-	-	-	-	-
Google GKE	✓	✓	✓	-	-	✓	✓	-	-
[18], [19], [23], [24]	✓	✓	-	✓	✓	-	✓	-	-
[27], [28]	✓	-	-	✓	-	-	-	-	-
[29], [30]	✓	-	-	-	-	-	-	-	-
[31]	✓	-	-	-	-	-	✓	-	-
I-Scaler	✓	✓	✓	✓	✓	✓	✓	✓	✓

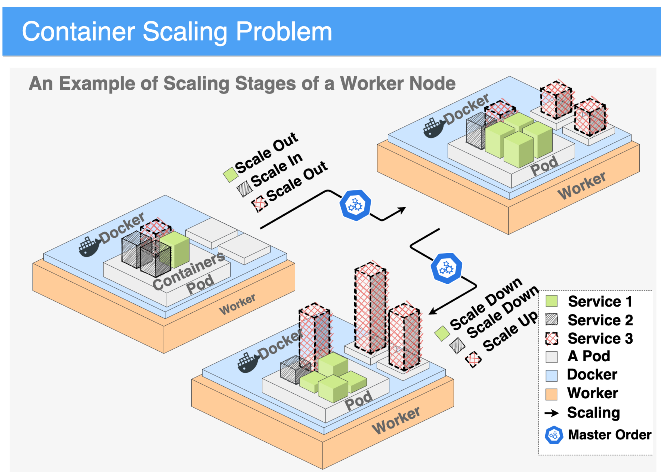


Fig. 1: Visual Representation of the Horizontal and Vertical Resource Scaling Problem

in and out. Scale-out means creating and placing copies of a micro-service in the cluster, while scale-in means removing placed instances. Horizontal scaling requires a service placement decision, which assigns or remove services to and from the servers based on preferred objectives. Besides, vertical scaling is composed of Scale-up and Scale-down methods, which adjust the CPU and Memory for a micro-service. Scaling in and down are very important for improving energy and resource consumption and offering more resources to be utilized by other applications running in the cluster. A better visualization of resource scaling and service placement is depicted in Fig. 1.

In this paper, we develop a multi-application scaling and placement solution, which can be integrated into any service-based clustering environment running containers and orchestration technologies, such as MEC. In reality, resource scaling is accompanied by many requirements and sub-problems, which render the decision challenging. For instance, multiple applications could run on the same cluster; therefore, multi-application resource horizontal and vertical scaling is essential to find a balance between the different applications. Besides, in case scaling is not proactive, the QoS and QoE are affected. For instance, in case of scaling is performed as demands occur, the user might encounter a service delay, which can be measured from the starting time of demands to the time when scaling is performed. Hence, proactive scaling of resources is mandatory. Knowing that available resources of worker nodes can change, a prediction of availability is necessary for proactive scaling. Aside from executing the scaling decision, the target hosting worker should be identified to remove or add instances. This is known as the service placement problem, where decisions are made based on many objectives that can be configured depending on the cluster's situation. In other words, the cluster might be situated at the edge; therefore, placing a service closer to the user is required.

IV. Architecture for Resource Provisioning in MEC Clusters

In this section, we present our architecture for integrating the IScaler technology in MEC clusters serving a 6G environment. This includes container-based clustering architectures managed using an orchestration technology. In this architecture, we apply changes to existing master nodes at the orchestration layer. These changes include the novel ISP module, which is responsible for performing intelligent scaling using IScaler and avoiding the challenges of using DRL.

A. *Architecture Overview*

For simplicity, we assume in this paper that Kubernetes clusters are used for scaling, which is dedicated to managing Docker containers. Kubernetes also offers a suitable environment for managing and scaling resources, as well as load balancing the tasks on running instances. As shown in Fig. 2, this architecture covers the common cases of running an MEC cluster that contains orchestrators and worker nodes. In the MEC layer, the orchestrator performs scaling through IScaler, and the MEC servers host services and execute scaling decision. Finally, the user layer tha generates requests.

The cluster manager node runs the necessary Kubernetes components for cluster and connection management. The master is responsible for adding and removing worker nodes from the cluster. Moreover, installing, removing, and performing physical scaling are done through the master controller. A connection to all workers is checked for ensuring healthy running services. Failure to reaching services results in rebooting or migration to other workers. Besides, the master node receives information about the loads of each worker to perform optimal load balancing. The information is stored in log files, which are used for IScaler learning. Utilizing these standard functionalities of the master node, we propose the integration

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

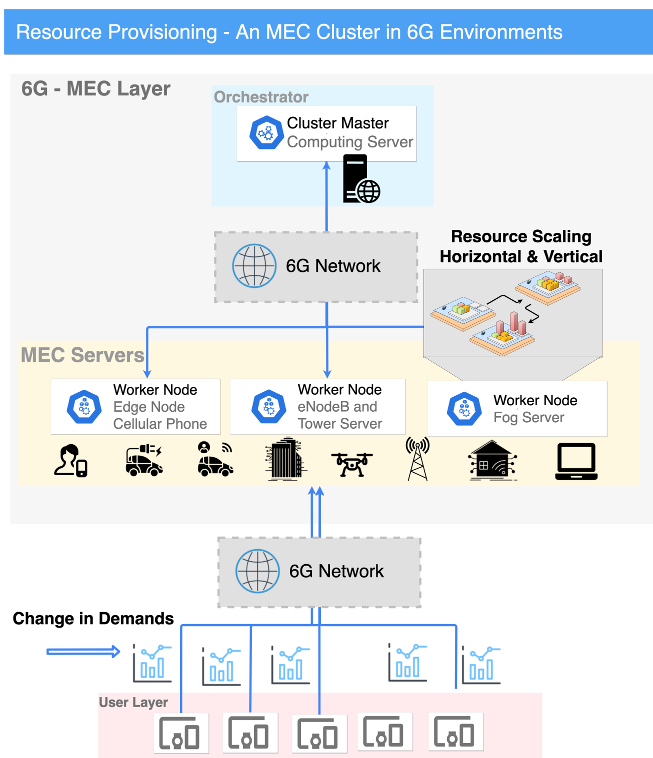


Fig. 2: Resource Provisioning Architecture for IoE Services Hosted by an MEC Cluster in a 6G Environment

of the ISP module described in Section IV-B2. Besides, the communications that happen between the master and worker nodes at the MEC layer of the 6G environment go through the 6G network.

The worker nodes in our architecture are running at the MEC layer. The nodes can resemble any computing device, ranging from a mobile phone to a powerful server or a base station compute engine. These devices run the required Kubernetes components to be able to join the cluster and communicate with the master. Worker nodes receive orders from the master to host services. These services run in the form of containers hosted inside pods. A worker is also responsible for sending periodic updates about the current status and the time of availability to the master. More importantly, the worker nodes host services for supporting the requests coming from the user layer. These requests arrive with different levels of demands that change over time and have to be supported. These demands are reflected on a load of worker nodes. Hence, the master's job is to load balance these requests and use IScaler to scale the available resources proactively using AI.

In our architecture, users requesting edge services through the 6G network can be any computing device that initiates requests and can range from small IoT devices to powerful computing servers. Using IScaler and in case the edge servers have enough available resources, users are guaranteed highly available applications and a satisfactory QoE with a high response rate and negligible delay.

B. Architecture Components

Resource scaling does not tolerate mistakes or sub-optimal decisions that directly affect the hosted applications by causing downtime and disrupting the QoS and QoE. IScaler utilizes model-free DRL; therefore, the agent learns the environment from scratch through interaction and trial and error. Henceforth, by using DRL, IScaler is subject to producing wrong decisions at the starting stages of learning, or when unseen patterns or new states are encountered and a model update is needed. To solve this issue, we propose in this section a novel architecture utilizing an optimizer combined with IScaler to cover possible errors during the learning phases. Also, our architecture offers a suitable environment for IScaler to learn using the collected application logs. A description of the orchestration layer's components is presented in Fig. 3 and described in the sequel.

1) CaaS Module: The Container as a Service (CaaS) module presents the different Kubernetes components that should be running on the master node. The cluster orchestrator is the manager for its cluster. Workers' initialization, management, and configuration happen through the cluster orchestration entity. Moreover, this entity is responsible for updating the logs that represent the worker nodes' status and load. Besides, the cluster controller component is the direct connection with the worker node, which distributes, scales, and organizes services following the instructions of the cluster orchestration entity [36].

2) The Intelligent Scaling and Placement (ISP): The intelligent scaling and placement sublayer is composed of IScaler, the Optimizer, and the Solution Switch. These components can be integrated into existing cluster orchestrators for performing resource scaling. IScaler is the DRL-based resource scaling solution that is responsible for proactively scaling computing resources and placing newly formed services on available servers. DRL solutions require time to learn by interacting with the environment in two cases: (1) learning the environment from scratch through trial and error, and (2) facing unseen patterns of services' demand or available resources in the studied data. These two factors cause the agent to make mistakes while performing the scaling decisions. To overcome this issue, we host a heuristic solution on the orchestrator as an alternate solution to replace IScaler while learning, and that confirms IScaler's correctness when making decisions on newly observed states or patterns. In other words, the Optimizer component is the bootstrapping tool for IScaler. The Solution Switch unit is utilized to check if IScaler's learning converges, to find the right time to switch between the Optimizer and IScaler solutions. This unit takes the output of IScaler and compares it with the one issued by the Optimizer. For simplicity, we use a threshold-based approach that counts and evaluates the correctness of the IScaler decision compared to heuristics. In case this count exceeds a predefined threshold, for instance, one hundred consecutive correct decisions, the orchestrator switches to using IScaler. It is also important to note that a heuristic solution cannot replace IScaler because: (1) a heuristic solution has to wait for demands to occur because it cannot take proactive decisions, and (2) heuristics

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

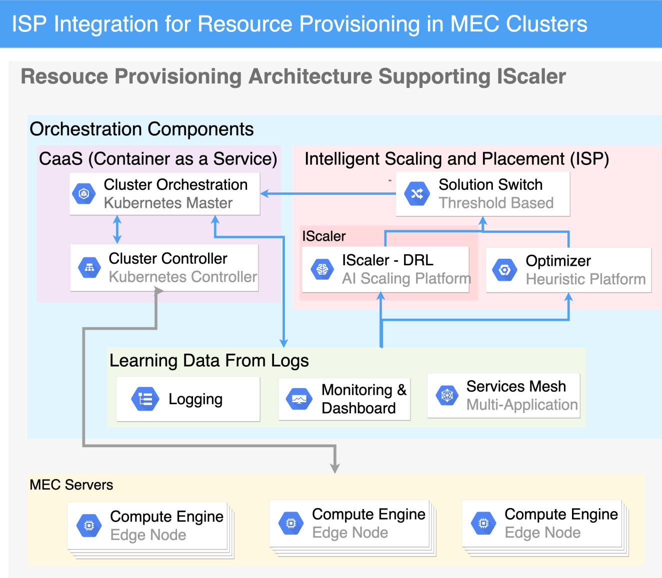


Fig. 3: MEC Architecture Components Embedding ISP for Enabling IScaler

cannot always guarantee a good solution.

3) Learning Data From Logs: Data utilized by the Optimizer and IScaler to learn and make decisions are provided by the Solution Switch module. This module keeps track of the current loads of each edge server and monitors the demands of hosted micro-services at the edge. The service mesh component is utilized by the Optimizer and IScaler to respect the connection between microservices. Despite that these data are used for learning, they can be used by the Solution Switch to monitor IScaler's performance to take further actions.

V. MDP Formulation for IScaler

IScaler agent utilizes a model-free DRL algorithm for learning. DRL takes as input a representation of the environment MDP model and tries to learn its dynamics or state transitions following some actions. In this section, we present a novel MDP formulation for performing proactive resource scaling and service placement while considering the change in users' demand and available resources. Our MDP design guarantees scalability by handling large inputs. In other words, IScaler is still able to perform fast learning in case of a large input, in addition to consuming less memory while learning.

A. Background

An MDP is a mathematical formulation for modeling sequential decisions in a stochastic environment and is the main framework applied to problems solved using RL. An MDP is characterized by the following tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{C}, \mathcal{Y})$. This tuple is a design choice that can affect the RL solution scalability and speed of convergence. $\mathcal{S} = \{s_1, s_2, \dots\}$ is the state space, i.e., the set of all states of the environment. Problems can have a finite or infinite number of states. $\mathcal{A} = \{a_1, a_2, \dots, a_l\}$ is the action space, i.e., the set of possible actions the agent can take at any given state. $\mathcal{P}r$ is the probability transition matrix which outputs for each state s the probability distribution for going to the next state s' when performing action a. When $\mathcal{P}r$ is given, model-based RL is used. However, in most real-life applications, $\mathcal{P}r$ is not known. In this case, model-free RL techniques are used to estimate it. $\mathcal{C}$ is the cost function which reflects the objectives of the agent. $\mathcal{C}$ takes the current state, action, and next state, and outputs a value to be minimized. Finally, $\mathcal{Y}$ is the discount factor, which is a decimal number $\in [0, 1]$ and is usually close to one. The main use of $\mathcal{Y}$ is to speed the convergence by discounting over the reward of the next states. In order to build a DRL solution for IScaler, we need to define these elements of the MDP tuple. In the sequel, we provide a novel design of each element.

B. State and Action Spaces

IScaler is a multi-application scaling solution, where every application has a set of services. Hence, we denote by $\mathcal{G} = \{G_1, G_2, \dots, G_g\}$ the set of applications of size g, which are represented by services. In addition, We denote by $\mathcal{E} = \{E_1, E_2, \dots, E_n\}$ the set of services of size n. A service $E_i \in \mathcal{E}$ is represented as: $E_i = [E_i^{cpu}, E_i^{mem}, E_i^{pri}, k]$, where $1 \leq i \leq n$ and E_i^{cpu}, E_i^{mem} are the CPU and memory requirements respectively. Moreover, E_i^{pri} is an integer representing the priority level over other services. When E_i^{pri} is high, E_i has a high priority to be considered for scaling and placement before other services with lower priority. Finally, k is the application index implying that $E_i \in G_k$. On the other hand, we denote $\mathcal{H} = \{H_1, H_2, \dots, H_m\}$ the set of available hosts of size m that are running the services in $\mathcal{E}$. Every host H_j is represented as $H_j = [H_j^{cpu}, H_j^{mem}, H_j^{dis}]$, where $1 \leq j \leq m$ and $H_j^{cpu}, H_j^{mem}, H_j^{dis}$ are the CPU and memory available and the distance of this host from the group of requesting users respectively. As highlighted in Section IV, the hosting cluster can run at the MEC layer. In this case, there are specific requirements for hosting services. For instance, service placement should consider minimizing the hosts distances from the area of the users. Consequently, the host distance feature is considered later as one of the objectives in the MDP cost function described in Section V.

In our state space, we represent the change in users demand and resource availability of each host in the cluster at different timestamp t. In each state, $q(t)$ represents the change in demand for different services, where $q(t)_i$ is a matrix of size $m \times 2$, which contains the average resource usage of CPU and memory of service E_i at t for every host in $\mathcal{H}$. The values in $q(t)$ are normalized to the total resource available on the hosts. Furthermore, we denote by $r(t)$ a matrix of size $m \times 2$ representing the normalized available resources of all hosts at t. $r(t)_j$ denotes the line j of $r(t)$ that represents the average resources for host H_j, i.e., $r(t)_j^{cpu}$ for CPU and $r(t)_j^{mem}$ for memory. Besides, available resources at a given state can be bounded by how much resources can service E_i use to scale. This boundary is set by a system administrator in order to leave space for other applications to scale in case the system is overloaded. To keep track of the latest scaling decisions, we denote by $p(t)$ the matrix of size $m \times n$ that stores these

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

decisions taken at each host for each service. Each element $p(t)_{i,j}$ contains the CPU and memory allocations represented as $p(t)_{i,j}^{cpu}$ and $p(t)_{i,j}^{mem}$ respectively. Thereafter, we represent a state s at t in our state space as follows:

$$s(t,i,j) = (q(t), r(t), p(t), i, j) \tag{1}$$

Following Equation 1, i and j are part of the state representation to denote the current service and host the agent is performing the decision for. The full $p(t), q(t), r(t)$ matrices are required in each state representation to make a combined decision while considering all services requirements and hosts availabilities. During t, the agent passes over all services and makes scaling decisions for each one separately. This reduces the action space and makes our MDP design scalable no matter the input size.

The action space in our MDP has a constant size which is very important for the model scalability. Every action a is composed of a list of two elements that hold the scaling decision of the CPU and memory for a given state. A scaling decision for CPU for instance is denoted as $a[0]$ and belongs to $\{-u, -1, 0, 1, u\}$. In this set, ± 1 denotes horizontal scaling, $\pm u$ is a decimal value that denotes vertical scaling, and 0 means no action is taken. It is important to emphasize that some actions are not feasible; however, if taken, the agent is punished using our cost function described in Section V-D.

C. States Transition and Model Dynamics

IScaler state design presented in Section V-B relies on $q(t)$ and $r(t)$, which are the applications' resource requirements and the available resource at the next timestep t respectively. The values of these lists are unknown and are hard to predict. In other words, $q(t)$ and $r(t)$ have a stochastic behavior which is based on the demand change of the user for an application and the change in resource usage of hosting servers in the cluster. Because these values are unknown, $\mathcal{P}r$ of our MDP is unknown. Henceforth, the RL algorithm that should be used for these environments is model-free. On the other hand, the state design entails the ability to perform scaling decisions for large clusters hosting several applications. To avoid blowing the action space, each state within a given timestep is divided into several steps.

Assuming that the current state is at t, the state representation is $(q(t), r(t), p(t), i^+, j^+)$. For instance, if the timestep is t, there are two loops of iterations defining the next states. The first loop considers fixing an application service and increasing j by one until passing over all the hosts and choosing the proper scaling action from $\mathcal{A}$. Once $j = m$, $j+$ becomes zero and i is increased by 1, which is denoted as i^+. Hence, $j^+ = j < m\ : j+1\ ?\ 0$. In addition, $i^+ = j = m\ : i+1\ ?\ i$, which means that $i+$ increases i by 1 in case $j = m$ and does not change i otherwise. Moreover, $p(t)$ at a state is updated by every scaling decision for the given i and j. It is important to note that for these internal iterations within a timestep, q and r are fixed until the agent moves to the next timestep. In this case, i, j, and $p(t)$ reset to zero, and a new q and r are observed by the agent.

D. Cost Function

Given the current state, the action taken, and the next state the agent results in, the cost function is calculated. When navigating in the state space, the goal of IScaler is to select the best action of the current state that results in the minimum cost. In other words, the correct selection of the action using the cost function C helps in forming an optimal policy for IScaler. In this section, we present four different objectives composing the cost function. These objectives are: (1) minimize the application load, (2) minimize the overload of the available resources, (3) minimize the containers priority cost, and (4) minimize the cost of other customizable objectives, such as minimizing the distance cost from the serving edge workers to actors. Taking an action moves the agent to the next timestep, i.e. from $t-1$ to t. A cost is represented as $C(s(t-1), a(t)|s(t))$. In the sequel, we present the mathematical formulation of each objective in the proposed cost function.

1) Minimize Application Load: The purpose of this objective function is to meet the load of different applications at the next timestep. Considering that the applications' loads are predicted, C_1 evaluates the scaling decision and compares the allocated resources to the ones required by each application. If the scaling decision underestimates the load, the cost returned is the difference between the actually required resources and the scaled ones. In case the demands are met for an application, the resource cost returned is zero. For this objective, we consider the cost of meeting the applications' resource requirements for both CPU and memory. Mathematically, C_1 of the CPU cost is represented in Equation 2.

$$C_1^{cpu}(t) = \frac{\sum_{i=1}^{n}(q(t)_i^{cpu} - \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu})}{\sum_{i=1}^{n} q(t)_i^{cpu}} \tag{2}$$

$$\text{such that } \forall i,\ \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu} < q(t)_i^{cpu}$$

where $q(t)_i^{cpu}$ is the CPU usage of service i and E_i^{cpu} is its CPU requirement. Otherwise, if $\exists i\ \ s.t.\ \ q(t)_i^{cpu} \leq \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu}$, the cost of this service is zero because the resource requirements for the application are met. We also divide the cost by $\sum_{i=1}^{n} q(t)_i^{cpu}$ for normalization. It is important to note that Equation 2 refers to CPU calculation, which is the same for memory calculation; however, we use $q(t)_i^{mem}$ and E_i^{mem} instead of $q(t)_i^{cpu}$ and E_i^{cpu}. Finally, $C_1(t) = C_1^{cpu}(t) + C_1^{mem}(t)$.

2) Minimize Available Resources Overload: In this objective function, the agent is punished for exceeding the use of available resources using the proactive scaling decision made. We denote C_2 as the cost of this objective. C_2 represents the resource overload cost by each application for the CPU and memory on each host. The CPU cost for this objective is represented mathematically in Equation 3.

$$C_2^{cpu}(t) = \frac{\sum_{j=1}^{m}\sum_{i=1}^{n}(p(t)_{j,i}^{cpu} \times E_i^{cpu}) - q(t)_i^{cpu}}{\sum_{j=1}^{m} r_j^{cpu}} \tag{3}$$

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

$$\text{such that } \forall j,\ \sum_{i=1}^{n}(p(t)_{j,i}^{cpu} \times E_i^{cpu}) > q(t)_i^{cpu}$$

In case the sum of scaled resources on at least one host underestimates available resource (i.e. $\exists j,\ \sum_{i=1}^{n}(p(t)_{j,i}^{cpu} < 0)$, the agent is punished for the action taken and a cost of k is returned for this objective, where $k > 1$. The same punishment applies when the scaling decision surpasses the available resources of any host (i.e. $\exists j,\ \sum_{i=1}^{n}(p(t)_{j,i}^{cpu} \times E_i^{cpu}) > r(t)_j^{cpu}$). Equation 3 applies if the usage surpasses the required resource load identified. Hence, the sum of the difference between the scaled resources of each application and the actual required resource load is returned. Otherwise, the cost is zero. In addition, A normalization factor of $\sum_{j=1}^{m} r_j^{cpu}$ is considered. Finally, Similar to C_1, $C_2(t) = C_2^{cpu}(t) + C_2^{mem}(t)$.

3) Priority Cost: A priority level is assigned to each service description. This value prioritizes the scaling of a service over others, which is important in the multi-application context and helps IScaler make efficient decisions. The cost of this objective is denoted as C_3. C_3^{cpu} is mathematically formulated in Equation 4 showing the CPU cost.

$$C_3^{cpu}(t) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(q(t)_i^{cpu} - p(t)_{j,i}^{cpu} \times E_i^{cpu}) \times E_i^{pri}}{\sum_{i=1}^{n} q(t)_i^{cpu} \times E_i^{cpu}} \quad (4)$$
$$\text{such that } \forall i,\ \sum_{j=1}^{m} p(t)_{j,i}^{cpu} \times E_i^{cpu} < q(t)_i^{cpu}$$

In case the resource loads of the application at the next timestep are met, or the service priority is zero, the cost of this service is zero. Otherwise, Equation 4 is applied to calculate the remaining amount of resources needed to meet the resource load requirements of that service. Finally, $C_3(t) = C_3^{cpu}(t) + C_3^{mem}(t)$.

4) Minimize Distance Cost: As highlighted earlier, the infrastructure admin can add custom objectives to our IScaler cost function. Using this custom objective, IScaler can adapt and produce the desired scaling actions following specific preferences related to the hosting environment. Supposing that the cluster where IScaler is deployed is hosted at the edge, one of the possible objectives to consider is minimizing the distance between the edge worker and the group of requesting users. Therefore, we present in Equation 5 a mathematical representation of C_4 for minimizing the total distance cost.

$$C_4(t) = \frac{\sum_{j=1}^{m} v(t)_j \times H_j^{dis}}{\sum_{j=1}^{m} H_j^{dis}} \quad (5)$$

where H_j^{dis} is the distance cost of host H_j, and $v(t)$ is a vector of size m and is calculated as follows: $\forall j,\ v(t)_j = 1$ if $\sum_{i=1}^{n} p(t)_{i,j} > 0$ and 0 otherwise. A normalization factor of $\sum_{j=1}^{m} H_j^{dis}$ is added.

Therefore, our cost function becomes:

$$\mathcal{C}((s(t-1), a(t))|s(t)) = \lambda_1 \times C_1(t) + \lambda_2 \times C_2(t) + \lambda_3 \times C_3(t) + \lambda_4 \times C_4(t) \quad (6)$$

where $\lambda \in [0,1]$ is a weight corresponding to each cost function given $\sum_{i=1}^{4} \lambda_i = 1$. These weights are tuned depending on the applications requirements and the nature of the cluster to give some cost functions more importance over the others, where the aim is to minimize $\mathcal{C}((s(t-1), a(t))|s(t))$.

VI. Intelligent Scaling and Placement (ISP)

A. IScaler using Deep Reinforcement Learning

The IScaler agent interacts with the environment for evaluating the placement action taken for each container. The agent executes actions for every state encountered and builds a strategy that adapts to the stochastic demands of users requesting services, as well as the change in available resources on worker nodes. The end goal of the agent is to learn the transition probability distribution from a state to all next states and find the optimal policy π^*, which takes as input a state and outputs the action that minimizes the future cost. In other words, π^* is a strategy or a set of actions the agent takes to minimize the cost. The future costs are discounted by γ, which controls the effect of future actions on past and current states, and helps the agent achieve faster convergence. let $\mathbb{C}(s(t-1), \pi|s(t))$ be the future discounted cost implied by choosing policy π at t that indicates selecting an action $a(t^{'})$, such that $t \leq t^{'} \leq \mathcal{T}$ where $\mathcal{T}$ is the final timestep of the episodes. $\mathbb{C}(s(t-1), \pi|s(t))$ is computed as follows:

$$\mathbb{C}(s(t-1), \pi) = \sum_{t^{'}=t}^{\mathcal{T}} \gamma^{t^{'}-t}\mathcal{C}(s(t^{'}-1), a(t^{'})|s(t^{'})) \quad (7)$$

We denote by $Q^*(s,a)$ the optimal action value function which minimizes the average expected cost for any selected strategy. It is expressed as follows:

$$Q^*(s,a) = \min_{\pi} \mathbb{E}[\mathbb{C}(s(t-1), \pi)] \quad (8)$$
$$\text{where } s(t-1) = s, a(t) = a$$

Let $[s..\mathcal{L}]$ be the chain of states from s to $\mathcal{L}$ linked by transitions using $\mathcal{P}r$. The optimal Q-function selects the action of the next state that minimizes the action value function following Equation 9:

$$Q^*(s,a) = \mathbb{E}_{s^{'} \in [s..\mathcal{L}]}[\mathcal{C} + \gamma \min_{a^{'}} Q(s^{'}, a^{'})] \quad (9)$$

where $\mathcal{C}$ is the immediate cost from Equation 6, and $\mathbb{E}_{s^{'} \in [s..\mathcal{L}]}$ is the expected value from the current state s to the last state $\mathcal{L}$ at $\mathcal{T}$. The basic form of RL is to find the optimal action value function using iterative updates following the Bellman equation. This update can be expressed as:

$$Q(s,a) := Q(s,a) + \alpha[\mathcal{C} + \gamma \min_{a^{'}} Q(s^{'}, a^{'})] \quad (10)$$

where α is the learning rate. In Equation 10, the update of the Q-function happens following the Q-learning algorithm [37]. All Q-values are stored in a table structure containing the list of states and actions. An exploration-exploitation trade-off aids the agent into interacting with the environment by covering the maximum number of possibilities, observing the cost signal, and updating the Q-values using Equation 10.

However, the use of tabular RL is not practical in our problem, where we have a large state space. The state-space can

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

grow with an increase in the number of containers and hosts to place. Thus, handling the whole table in memory, trying to cover all possible actions for every state, and updating the Q-values for all of them is computationally very expensive. Such an implementation is time-consuming and makes any tabular RL agent diverges [38]. As a solution, learning the optimal Q-values can be retrieved from some adjustable weights denoted as θ. These weights get updated using gradient descent to update the weights downwards towards the direction of the gradient for minimizing the error of the calculated Q-values for every iteration. The common form of approximation is the linear function approximation, which generalizes the environment through its weight, where the Q-function becomes close to the optimal Q^* having $Q^*(s,a) \approx Q(s,a,\theta)$.

Given the advantages of a linear approximation to overcome the tabular learning limitations, these models will not be able to generalize well when the model complexity and state spaces increase. Here comes the advantage of using non-linear approximations such as Deep Neural Network (DNN) to approximate the environment, giving the agent the power of Deep Learning (DL) to update its weights, where learning can be customized [39]. The Deep Q-Network (DQN) algorithm has the advantage of merging the concepts of RL and DL. Henceforth and after experimenting with the different linear approximation approaches for building our IScaler agent, including Temporal Difference TD(0) and TD(λ) [40], DQN outperforms the other approximation methods. Algorithm 1 provides a pseudo-code of our IScaler learning algorithm, which benefits from the advancement in DQN.

As illustrated in Algorithm 1, we start by creating a multi-layer perceptron for the source model used for calculating the state action-value function Q using its weights θ. The input to the model is a transition sample, and the output is a single neuron with linear activation. A target multi-layer perceptron is created, which is a copy of the source model. We denote by θ^- the weights of the target model, which are a copy of θ in the initialization phase (line 1). We then initialize a replay buffer D of size $G = 1000$, which stores the transition containing the current state, the action taken, the cost retrieved, and the next state-observed (line 2).

The learning starts by initializing a random state $s(t)$ at the beginning of every episode (line 5). X episodes are played for learning. X varies depending on the input size for the test case. Each episode is bounded by $\mathcal{T}$ learning steps. Every step starts by deciding on the action taken for the current state. We implement this decision by following the ϵ-greedy policy, which is essential for achieving a trade-off between exploration and exploitation. In ϵ-greedy, we set ϵ to be a variable that decays over time. For instance, $\epsilon = \frac{B1}{B2+NI}$ decreases as the number of iterations NI increases, where $B1$ and $B2$ are two constants such that $B1 < B2$. We then generate a random value of w between zero and one. If $1 - \epsilon > w$, we select an action randomly from the action space (lines 8-9). This is known as an exploration iteration for the agent. Otherwise, the action having the maximum of Q-value in the source model is selected (lines 10-11). This is known as the exploitation iteration.

After taking the action, the agent observes the service demands and available resources after the service placement is updated. This then allows the agent to calculate the cost $\mathcal{C}(s(t), a(t+1)|s(t+1))$ using Equation 6. After forming the next state $s(t+1)$, a transition is stored in the replay buffer (lines 13-18). Because updating the model online as data comes causes instability, data are stored in the replay buffer. Samples from these data, of size $Y = 50$, are extracted randomly and uniformly to form the mini-batch dataset for the model to train and break the problem of correlation between sequences of actions (line 19). As mentioned previously, the source weights are stored in the target model. This is vital to improve the source model learning stability. The source model adjusts θ of Q-function by using the predicted Q-values of

Algorithm 1: IScaler Algorithm Using DQN

```
1  Build a Multi-Layer Perceptron as source model to
     calculate Q and randomly initialize its weights θ;
2  Build a target model for Q with weights θ⁻ which are
     a copy of θ;
3  Initialize replay buffer D to capacity G;
4  while episode X do
5      Initialize a random state s(t);
6      Reset t;
7      while t < T do
           /* following ε-greedy policy */
8          if Random Selection then
9              select a(t+1) randomly from feasible
                 actions;
10         else
11             a(t+1) = max_a Q(s(t), a, θ);
12         end
13         Update p(t+1), observe q(t+1) and r(t+1);
14         Update j to j⁺; // if applicable
15         Update i to i⁺; // if applicable
16         Build s(t+1);
17         Calculate C(s(t), a(t+1)|s(t+1)) using
             Equation 6;
18         Store [s(t); a(t+1); C(s(t), a(t+1)|s(t+1));
             s(t+1)] in D;
19         Select random mini-batch transition of size Y
             from D;
20         for k in length(mini-batch) do
21             y_k = C_k + γ min_a' Q(s_{k+1}, a', θ⁻);
22         end
23         Update θ using gradient descent towards
             minimizing the loss: (y − Q(s, a, θ))² for
             every transition;
24         if length(D) > G then
25             Pop out the oldest element in D;
26         end
27         Every Z steps, copy θ into θ⁻;
28         Update the current state to s(t+1);
29         Increment t;
30     end
31     Increment Episode;
32 end
```

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

the target model as labels (lines 20-22). This, in turn, builds a supervised learning context with a fixed dataset and labels on which to train. In our implementation of IScaler-based DQN, loss functions are inferred and calculated for every iteration using the mean squared error loss (line 23). This loss is back-propagated to the neurons using the gradient-descent towards minimizing the loss to get a better estimate of Q (line 23). To preserve the RL concept for allowing the model to keep on improving the Q-function as new data come, the replay buffers D keeps on updating slowly by removing the oldest transitions at every iteration when the buffer is full (lines 24-26). On the other hand, the weights for the target model θ^{-} keeps on updating after $Z = 500$ (line 27).

B. Optimizer

An Evolutionary Memetic Algorithm (MA) is an extension of the genetic algorithm that includes a local search. The local search allows the heuristic solution to minimize the chance of halting in a local optimal and offers faster convergence to a near-optimal solution. The MA is used in our previous research work for resource management [6], [41].

In the context of resource scaling and management, we utilize the Optimizer to support IScaler in producing efficient decisions. The implementation of the Optimizer is based on the MA solution. In terms of the formulation of the MA, the inputs to the algorithm are: the set of services $\mathcal{E}$, the set of hosts $\mathcal{H}$, the set of resource usage of each application q after the occurrence of the demands, and the set of currently available resources r. The output of the MA is a two-dimensional matrix of size $m \times n$ representing the scaling value of each container on every host. An element in the output matrix is a value in $[0, \mathcal{G}_{max}]$ where $\mathcal{G}_{max}$ denotes the maximum value a container can be scaled to. Moreover, obviously, our MA solution for the Optimizer uses the same cost function presented in Equation 6. This allows the Solution Switch to be able to compare the output of the Optimizer and IScaler. More details about the MA implementation can be found in [6].

VII. Experiments and Evaluations

In this section, we describe the experimental setup and elaborate on the different experiments conducted to show the efficiency of the proposed IScaler in different contexts, the advantage of utilizing the Optimizer within ISP, and finally a comparison with a recent existing scaling solution. In brief, the objectives are:

- Study IScaler DRL model convergence in a multi-application context, in addition to studying the efficiency of the decisions made towards resource provisioning.
- Highlight the advantage of using our Optimizer on the orchestrator during the learning phase of IScaler.
- Compare the performance of IScaler to a model-based RL algorithm for edge computing environments [18].

A. Experiment Setup

To meet the objectives of the experiment, we implemented a DRL algorithm based on the proposed MDP design for building IScaler, a model-based RL algorithm called Dyna-Q [18] for a comparative study with IScaler, the MA for simulating the behavior of the Optimizer, and finally the Solution Switch. The experiments were executed on a Windows 10 machine having a Core-i7 (12 CPUs), 32GB of RAM, and an Nvidia Quadro P620 graphic card for GPU training. The programming language used is Python V3.7, and we relied on the Tensorflow library for the implementations of the source and target deep learning models of IScaler [42]. The source and target networks for the DRL implementation are deep neural networks that consist of four layers having 32, 16, 8, and 1 neuron, respectively. The activation function on the hidden layers is ReLU, while we are using a linear activation on the output. The networks are configured to use the RMS optimizer, the Huber loss, a learning rate of 0.001, and a batch size of 60.

Fig. 4: GCT Services Resource Demands

The conducted experiments are based on simulations on the Google Cluster Usage Traces v3 2019 dataset (GCT) [26]. In this dataset, google physical machines are used and grouped into cells (clusters) having different resources allocated and available. Moreover, jobs refer to users' requests to execute a certain task on the cluster. GCT provides the data describing each machine in the cell, the resources allocated and available, the jobs to be executed on each machine, and the required resources in terms of CPU and memory. In our work, we map the cell to a Kubernetes cluster where IScaler, the Optimizer, and the Solution Switch are running. Machines in the cells resemble the worker nodes of our clusters. Furthermore, the jobs correspond to the containers to be scheduled in the cluster and scaled using our solution.

Data pre-processing, cleaning and visualization are performed on the dataset. For simplicity, three jobs/containers and three machines/worker nodes are selected from the dataset and visualized in Figs. 4 and 5. In Fig. 4, we show the average change in CPU demands by three different services over multiple samples from the dataset. In this figure, the curve drops imply a decrease in resource demands and therefore fewer requests from users arrive. In contrast, the curve reaches higher values when the demands of users increase. On the other hand, Fig. 5 illustrates the change of the average available resources for three different hosts sampled from the dataset. These figures demonstrate a real-life scenario for the change in demands and offered resources in the cluster. We benefit from this data in our experiment to show the ability of IScaler to adapt to these changes and perform efficient scaling and service placement decisions. For Simplicity, we display the CPU usage in the results instead of both CPU and Memory.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

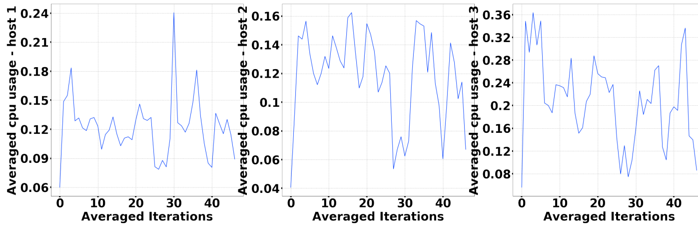


Fig. 5: GCT Hosts Available Resources

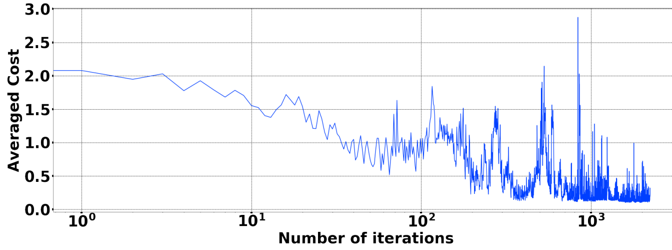
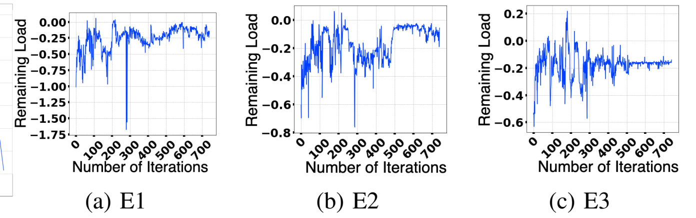
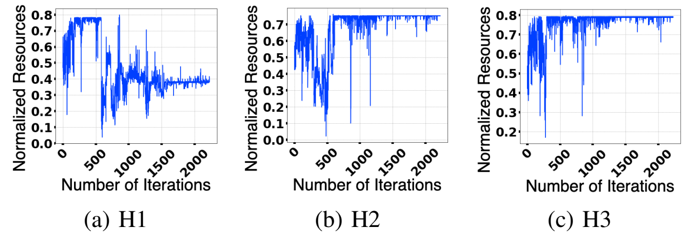


Fig. 6: IScaler Convergence



(a) E1 (b) E2 (c) E3

Fig. 7: The Difference Between Actual Demands and Offered Resources for Each Service



(a) H1 (b) H2 (c) H3

Fig. 8: Remaining Available Resources of Each Host

B. Multi-Application Model Convergence

In this part, we experiment with the performance of IScaler in a multi-application setting using the GCT dataset. In particular, we use the three samples of services and hosts described in Section VII-A. Services are presented by $\{E_1, E_2, E_3\}$, and hosts by $\{H_1, H_2, H_3\}$. Following the objectives of our cost function described in Section V-D, we assign to each service a priority level, and for each host a distance value. The distance value represents the distance between the host location (longitude and latitude), and the central point between a group of users. The priority levels are assigned as follows: $E_1^{pri} = 1,\ E_2^{pri} = 3,\ E_3^{pri} = 2$. Moreover, the distance value assigned to each host are: $H_1^{dis} = 10\ H_2^{dis} = 20\ H_3^{dis} = 30$. Besides, we assign different weights to each objective of the cost described in Section V-D. The weights assigned are: $\lambda_1 = 0.2$, $\lambda_2 = 0.4$, $\lambda_3 = 0.2$, $\lambda_4 = 0.2$. Consequently, the objective of minimizing the resource load of the application has more influence on the decision of the agent.

Following two million iterations of learning for IScaler using data of demands and resource availability fed from the GCT dataset, the model can converge with respect to the cost value produced for every decision. The long time of convergence is interpreted by the stochastic nature of demands and available resources on the GCT dataset as shown in Figs. 4 and 5. In Fig. 6, we show the convergence of our proposed DRL solution. In this figure, we plot the variation of the average cost value with respect to the average number of iterations, which are considered epochs. This graph is displayed on a logarithmic scale for better visualization of the agent performance.

In addition, we study the efficiency of decisions made during the learning and after the convergence with respect to every objective of our cost function. In Fig. 7, we show the amount of CPU resource load that is proactively prepared for each service after scaling. In each graph, we plot the averaged difference between the actually required resource and the offered resources in the cluster with respect to the averaged number of iterations. This means that a closer value to zero means exactly the required demands are offered. On the other hand, a negative value indicates that the amount of offered resources exceeds the actual requirements of each application.

As shown in the results of each figure, the amount of offered resources is most of the time larger than the required ones at the beginning of learning. This is because $\lambda_2 = 0.4 > \lambda_1 = 0.2$. Thus, meeting the amount of required resources has more impact compared to using available resources. Besides, the amount of utilized resources is approaching zero in each graph as the agent converges. In the end, the agent is able to learn the optimal resource allocation decisions for each service. More importantly, the resource of E_2 is exactly met at each iteration due to the high priority score.

Available resources change over time, thus it is important to check if IScaler is utilizing more than the available resources on each host, which should be avoided. In Fig. 8, we show the averaged difference between the utilized resources by IScaler and the available resources on each host. A value of zero means that IScaler is proactively using the exact amount of available resource on this host, while a positive value indicates the amount of remaining available resources IScaler can use.

As shown in Fig. 8, IScaler is able to converge while respecting the change of available resources on each host. It is important to note that the available resources on H_1 are more utilized compared to other hosts because H_1 has the shortest distance to the user. Moreover, as shown in the results of each figure, IScaler is not utilizing the full available resources on each host, therefore respecting the first objective in our cost function to minimize the amount of utilized resources.

In summary, IScaler is capable of performing efficient scaling decisions by meeting the load requirements for each application, more importantly, the ones with high priority, and respecting the amount of available resources for each host.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management

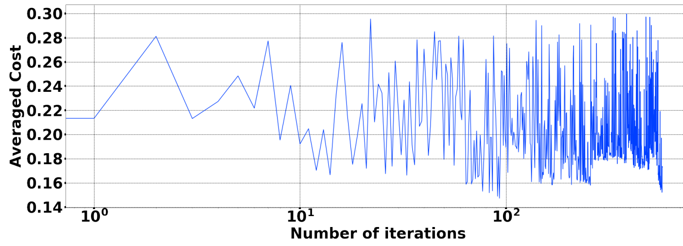
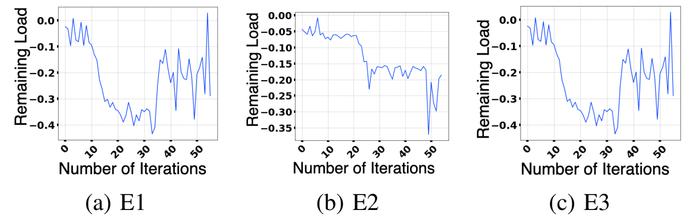
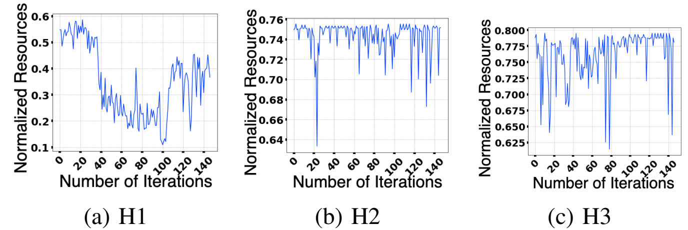


Fig. 9: ISP Performance



(a) E1 (b) E2 (c) E3

Fig. 10: Resource Load for Each Service



(a) H1 (b) H2 (c) H3

Fig. 11: Remaining Available Resources of Each Host

C. ISP Performance

As shown in the results of Figs. 6, 7, and 8, the agent is performing decisions that result in high costs on the environment. The high cost is reflected in shortening the applications' availability through unbalanced resource utilization and incorrect scaling of containers. This behavior of a DRL agent can occur in two cases. First, the agent is in the first stages of learning. Second, the agent is facing an unprecedented change in the environment that requires a model update. In both cases, the Optimizer can intervene to perform the scaling decision until IScaler develops a better model.

In order to experiment with the advantage of using the Optimizer, we simulate the behavior of combining the Solution Switch, the Optimizer, and the IScaler. While IScaler starts learning from scratch, we use the same setting and input of the previous experiment. The results of the averaged cost function using ISP with respect to an averaged number of iterations are shown in Fig. 9. As shown in this figure, the cost of the decisions made is in the range between 0.14 and 0.3, including the first iterations when IScaler is making inaccurate decisions. This explains the importance of using the Optimizer for replacing IScaler. In this experiment, we queue the results of the decisions made by IScaler and the Optimizer. After every 100 iterations, the Solution Switch evaluates both decisions to decide on the right solution to use. After $300,000$ iterations, the Solution Switch silently shifts from using the Optimizer to IScaler for proactive decisions. As shown in the graph, there are no jumps outside the range of $[0.14, 0.3]$ of cost because the model converges. One limitation remains when using the Optimizer, which is the inability of performing proactive decisions. Therefore, the scaling decision is made after the demands occur.

Furthermore, we study the impact of using the Optimizer on improving the quality of the decision to meet each of the objectives of our cost function described in Section V-D. Therefore, Figs. 10 and 11 present the amount of resource load met for each application and the utilization of available resources on each host, respectively.

As shown in the results of Fig. 10, the required resources for each application are always met. Besides, the CPU resource load of each service has a negative value sometimes. This implies that services are assigned more resources compared to the needed ones. The main reason behind this behavior is that meeting the resource load has more importance over minimizing the resources utilized on hosts ($\lambda_1 < \lambda_2$).

On the other hand, as shown in the results of Fig. 11, the available resources for H_2 and H_3 are less utilized by IScaler compared to H_1. The main reason is that the shortest distance from the users is H_1. Therefore, the IScaler decision respects the fourth objective of our MDP cost function to minimize the distance from users.

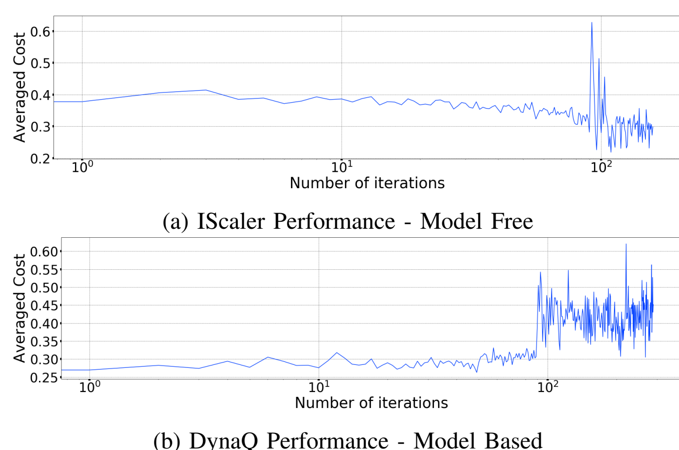
D. IScaler v.s. Model-Based Scaling

A recent literature work proposed a horizontal and vertical resource scaling for a single application using model-based reinforcement learning [18]. Despite that the service placement solution of that work is based on a heuristic solution, we compare in this section the performance of model-based RL to IScaler for a single application. Therefore, we replicate in this experiment the Dyna-Q model-based algorithm proposed in [18]. Some adjustments are applied to the state space of the Dyna-Q model to perform a fair comparison with IScaler. For instance, the features of service placement and the representation of the change of available resources are applied. Dyna-Q solution uses tabular Q-learning and estimates the probability transition matrix $\mathcal{P}r$ for learning the dynamics of the environment. Despite that estimating $\mathcal{P}r$ requires a lot of computation and sometimes is not practical in the case of large input spaces, the main objective of this experiment is to compare the behavior of Dyna-Q and IScaler when a change to the environment occurs. For this purpose, service E_1 is selected for scaling in both solutions on the three hosts. After multiple episodes of learning for IScaler, and one iteration for extracting $\mathcal{P}r$ for Dyna-Q, a major drop in demands is manually provoked in both environments. In order to compare the performance of each agent, the results of the averaged cost value are presented in Fig. 12b.

As shown in the results of Fig. 12b, the errors at the first stages of the decision making are negligible for the model-based Dyna-Q compared to IScaler performance in Fig. 12a. This is obvious because the dynamics of the environment are known for Dyna-Q. However, once unprecedented change happens in the resource demands of the application, the errors

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2021.3066625, IEEE Transactions on Network and Service Management



(a) IScaler Performance - Model Free

(b) DynaQ Performance - Model Based

Fig. 12: IScaler Performance vs Dyna-Q

for Dyna-Q jump, as shown in the first graph of Fig. 12b. This high error remains until $\mathcal{P}r$ of Dyna-Q is updated with new samples of data, making it impractical in such a dynamic environment. On the other hand, it is noticeable that IScaler is able to re-adjust the model, adapt to the environment change, and converge again in minimal time. IScaler uses model-free DRL for approximating $\mathcal{P}r$. This approximation dynamically changes with respect to changes in the environment that are interpreted through the reward signal.

VIII. Conclusion

Heading towards the development, hosting, and management of the new generation of services that are supported by 5G and 6G requires, there is a need for massive availability of computing resources, which is offered by the MEC. Due to the limitation of MEC available resources, dynamic resource management of multiple applications on the MEC infrastructure has been identified as one of the main challenges for the future of cellular networks. Therefore, we propose in this paper IScaler. IScaler is a DRL-based multi-applications resource scaling and service placement solution capable of overcoming the existing challenges of the dynamic environment with a stochastic change in demands to execute efficient decisions. Furthermore, adopting a DRL-based solution in 5G or 6G networks is very costly because of the errors the agent can make and the time required to learn. Thus, we propose an ISP module, which consists of IScaler, Optimizer, and Solution Switch. Through a series of experiments using the GCT dataset, we illustrated the efficiency of ISP decisions in (1) performing proactive intelligent multi-application scaling and placement decisions, (2) using the Optimizer during IScaler's model changes, and (3) demonstrating the ability of IScaler to outperform existing model-based scaling solutions.

References

[1] R. Vannithamby and S. Talwar, *Towards 5G: Applications, requirements and candidate technologies*. John Wiley & Sons, 2017.

[2] W. Saad, M. Bennis, and M. Chen, "A vision of 6g wireless systems: Applications, trends, technologies, and open research problems," *IEEE network*, vol. 34, no. 3, pp. 134–142, 2019.

[3] H. Yang, A. Alphones, Z. Xiong, D. Niyato, J. Zhao, and K. Wu, "Artificial-intelligence-enabled intelligent 6g networks," *IEEE Network*, vol. 34, no. 6, pp. 272–280, 2020.

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[5] J. Cao, W. Feng, N. Ge, and J. Lu, "Delay characterization of mobile edge computing for 6g time-sensitive services," *IEEE Internet of Things Journal*, 2020.

[6] H. Sami and A. Mourad, "Dynamic on-demand fog formation offering on-the-fly iot service deployment," *IEEE Transactions on Network and Service Management*, 2020.

[7] L. Gavrilovska, V. Rakovic, and D. Denkovski, "Aspects of resource scaling in 5g-mec: Technologies and opportunities," in *2018 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2018, pp. 1–6.

[8] M. Giordani, M. Polese, M. Mezzavilla, S. Rangan, and M. Zorzi, "Toward 6g networks: Use cases and technologies," *IEEE Communications Magazine*, vol. 58, no. 3, pp. 55–61, 2020.

[9] H. A. Alameddine, S. Sharafeddine, S. Sebbah, S. Ayoubi, and C. Assi, "Dynamic task offloading and scheduling for low-latency iot services in multi-access edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 668–682, 2019.

[10] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[11] S. A. Rahman, A. Mourad, M. El Barachi, and W. Al Orabi, "A novel on-demand vehicular sensing framework for traffic condition monitoring," *Vehicular Communications*, vol. 12, pp. 165–178, 2018.

[12] N. Moati, H. Otrok, A. Mourad, and J.-M. Robert, "Reputation-based cooperative detection model of selfish nodes in cluster-based qos-olsr protocol," *Wireless personal communications*, vol. 75, no. 3, pp. 1747–1768, 2014.

[13] A. A. Abdallah, S. S. Saab, and Z. M. Kassas, "A machine learning approach for localization in cellular environments," in *2018 IEEE/ION Position, Location and Navigation Symposium (PLANS)*, 2018, pp. 1223–1227.

[14] W. Fawaz, R. Atallah, C. Assi, and M. Khabbaz, "Unmanned aerial vehicles as store-carry-forward nodes for vehicular networks," *IEEE Access*, vol. 5, pp. 23 710–23 718, 2017.

[15] W. Fawaz, "Effect of non-cooperative vehicles on path connectivity in vehicular networks: A theoretical analysis and uav-based remedy," *Vehicular Communications*, vol. 11, pp. 12–19, 2018.

[16] D. Vohra, *Kubernetes microservices with Docker*. Apress, 2016.

[17] D. M. Gutierrez-Estevez, M. Gramaglia, A. De Domenico, N. Di Pietro, S. Khatibi, K. Shah, D. Tsolkas, P. Arnold, and P. Serrano, "The path towards resource elasticity for 5g network architecture," in *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE, 2018, pp. 214–219.

[18] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, 2020.

[19] J. B. Benifa and D. Dejey, "Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Mobile Networks and Applications*, vol. 24, no. 4, pp. 1348–1363, 2019.

[20] N. Kherraf, H. A. Alameddine, S. Sharafeddine, C. M. Assi, and A. Ghrayeb, "Optimized provisioning of edge computing resources with heterogeneous workload in iot networks," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 459–474, 2019.

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

[22] K. B. Letaief, W. Chen, Y. Shi, J. Zhang, and Y.-J. A. Zhang, "The roadmap to 6g: Ai empowered wireless networks," *IEEE Communications Magazine*, vol. 57, no. 8, pp. 84–90, 2019.

[23] C. Bitsakos, I. Konstantinou, and N. Koziris, "Derp: A deep reinforcement learning cloud system for elastic resource provisioning," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2018, pp. 21–29.

[24] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 64–73.

[25] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Fscaler: Automatic resource scaling of containers in fog clusters using reinforcement

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.

learning," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2020, pp. 1824–1829.

[26] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[27] J. Kumar, A. K. Singh, and R. Buyya, "Self directed learning based workload forecasting model for cloud resource management," *Information Sciences*, vol. 543, pp. 345–366, 2020.

[28] I. K. Kim, W. Wang, Y. Qi, and M. Humphrey, "Forecasting cloud application workloads with cloudinsight for predictive resource management," *IEEE Transactions on Cloud Computing*, 2020.

[29] Z. A. Al-Sharif, Y. Jararweh, A. Al-Dahoud, and L. M. Alawneh, "Accrs: autonomic based cloud computing resource scaling," *Cluster Computing*, vol. 20, no. 3, pp. 2479–2488, 2017.

[30] C. Li, H. Sun, Y. Chen, and Y. Luo, "Edge cloud resource expansion and shrinkage based on workload for minimizing the cost," *Future Generation Computer Systems*, vol. 101, pp. 327–340, 2019.

[31] M. Scarpiniti, E. Baccarelli, P. G. V. Naranjo, and A. Uncini, "Energy performance of heuristics and meta-heuristics for real-time joint resource scaling and consolidation in virtualized networked data centers," *The Journal of Supercomputing*, vol. 74, no. 5, pp. 2161–2198, 2018.

[32] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[33] R. Li, Z. Zhao, Q. Sun, I. Chih-Lin, C. Yang, X. Chen, M. Zhao, and H. Zhang, "Deep reinforcement learning for resource management in network slicing," *IEEE Access*, vol. 6, pp. 74 429–74 441, 2018.

[34] Y. Z. Helin Yang, Z. Xiong, J. Zhao, D. Niyato, K.-Y. Lam, and Q. Wu, "Deep reinforcement learning based intelligent reflecting surface for secure wireless communications."

[35] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis, "Optimal and scalable caching for 5g using reinforcement learning of space-time popularities," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 180–190, 2017.

[36] P. Farhat, H. Sami, and A. Mourad, "Reinforcement r-learning model for time scheduling of on-demand fog placement," *The Journal of Supercomputing*, vol. 76, no. 1, pp. 388–410, 2020.

[37] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

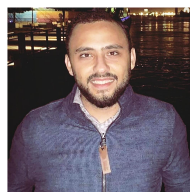
[38] X. Xu, L. Zuo, and Z. Huang, "Reinforcement learning algorithms with function approximation: Recent advances and applications," *Information Sciences*, vol. 261, pp. 1–31, 2014.

[39] S. S. Saab and D. Shen, "Multidimensional gains for stochastic approximation," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 5, pp. 1602–1615, 2019.

[40] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[41] H. Sami, A. Mourad, and W. El-Hajj, "Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 778–790, 2020.

[42] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.



Hani Sami is currently pursuing his Ph.D. at Concordia University, Institute for information Systems Engineering (CIISE). He received his M.Sc. degree in Computer Science from the American University of Beirut and completed his B.S. and worked as Research Assistant at the Lebanese American University. The topics of his research are Fog Computing, Vehicular Fog Computing, and Reinforcement Learning. He is a reviewer of several prestigious conferences and journals.

Hadi Otrok holds an associate professor position in the department of Electrical Engineering and Computer Science (EECS) at Khalifa University of Science and Technology, an affiliate associate professor in the Concordia Institute for Information Systems Engineering at Concordia University, Montreal, Canada, and an affiliate associate professor in the electrical department at Ecole de Technologie Superieure (ETS), Montreal, Canada. He received his Ph.D. in ECE from Concordia University. He is a senior member at IEEE, and associate editor at: IEEE TNSM, Ad-hoc networks (Elsevier), and IEEE Network. He served as associate editor at IEEE communications letters. He co-chaired several committees at various IEEE conferences. His research interests include: Blockchain, reinforcement learning, Federated Learning, crowd sensing and sourcing, ad hoc networks, and cloud and fog security.

Jamal Bentahar received the Ph.D. degree in computer science and software engineering from Laval University, Canada, in 2005. He is a Professor with Concordia Institute for Information Systems Engineering, Concordia University, Canada. From 2005 to 2006, he was a Postdoctoral Fellow with Laval University, and then NSERC Postdoctoral Fellow at Simon Fraser University, Canada. He is an NSERC Co-Chair for Discovery Grant for Computer Science (2016-2018). His research interests include the areas of computational logics, model checking, multi-agent systems, service computing, game theory, and software engineering.

Azzam Mourad received his M.Sc. in CS from Laval University, Canada (2003) and Ph.D. in ECE from Concordia University, Canada (2008). He is currently an associate professor of computer science with the Lebanese American University and an affiliate associate professor with the Software Engineering and IT Department, Ecole de Technologie Superieure (ETS), Montreal, Canada. He published more than 100 papers in international journal and conferences on Security, Network and Service Optimization and Management targeting IoT, Cloud/Fog/Edge Computing, Vehicular and Mobile Networks, and Federated Learning. He has served/serves as an associate editor for IEEE Transaction on Network and Service Management, IEEE Network, IEEE Open Journal of the Communications Society, IET Quantum Communication, and IEEE Communications Letters, the General Chair of IWCMC2020, the General Co-Chair of WiMob2016, and the Track Chair, a TPC member, and a reviewer for several prestigious journals and conferences. He is an IEEE senior member.

1932-4537 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Bibliothèque ÉTS. Downloaded on June 25,2021 at 07:34:24 UTC from IEEE Xplore. Restrictions apply.