FScaler: Automatic Resource Scaling of Containers in Fog Clusters Using Reinforcement Learning

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

***Abstract*—Several studies leverage fog computing as a solution to overcome cloud delays, including computation, network, and data storage. Along with the increase in demands for computing resources in fog infrastructures, heterogeneous fog devices are used towards forming highly available clusters. Existing approaches support the use of heterogeneous fogs and enable dynamic updates and management of services through containerization and orchestration technologies. However, none of the existing works proposed a proactive solution to horizontally scale these resources based on the IoT workload fluctuations, in addition to deciding on proper placement of the scaled instances on fogs with minimal cost on the fly. An effective scaling results in improving the response time and avoid service instability on fog devices. Therefore, we propose in this work FScaler, a reinforcement learning agent that horizontally scales container's instances after studying user's demands, and schedules the placement of newly created instances based on defined cost functions after studying the change in resources availability. The environment of FScaler is modeled as an MDP to be solved by any RL algorithm. For this work, we study the efficiency of our MDP formulation by solving the problem using SARSA. Promising results are shown through testing using a real-life dataset presenting the variation of user's demands of a particular service and the change in resource availability over time.**

***Index Terms*—Fog Computing, Horizontal Scaling, Service Placement, Reinforcement Learning, Container, Kubernetes**

I. Introduction

Fog computing is utilized to work out the limitations of distant clouds affecting IoT devices in terms of networking, computation, and data storage. Fog can be any computing device located close to the user. The creation of these fogs can be static [1] or dynamic [2], depending on the environment it is serving. Furthermore, virtual machines (VM) were used to facilitate service hosting as a flexible solution supporting any operating system and coping with the rise of fog computing. On the other hand, the increase in the number of IoT devices drain fog resources with service requests and raise the need for dynamically placing and updating fog services. Therefore, researchers deviate from using VMs to using containers to meet these demands [3]. Containers are proven to be more lightweight compared to VMs because they use the device's operating system rather than a new copy of it. More importantly, it is also more flexible to manage a large number of containers and resources in clusters of fogs using orchestrators. Kubernetes [4] is an example of a container orchestration tool capable of (1) monitoring the health of containers, (2) load balancing requests on multiple instances to avoid service overload and instability, and (3) managing the cluster resources.

To fulfill the increasing demands and requests for services hosted on fog devices, dynamic on-demand horizontal scaling of containers resources is imperative. Horizontal scaling is the act of adding and/or removing instances of containers to meet an acceptable response rate for users, or to clean resources to be consumed by other applications. A challenging factor in such decisions is the fact that fog resources' are heterogeneous and can change over time depending on the demands of other applications. This necessitates an automated approach that predicts the demand of groups of users to act proactively and scale the required number of instances. This approach must also decide upon the new instances placements inside the orchestrated fog clusters based on the changing available resources in order not to affect other running applications.

Meanwhile, Reinforcement Learning (RL), a category of machine learning, utilizes intelligent agents to build knowledge, take actions, and adapt to the changes in its environment through a load balance between exploration and exploitation. The agent aims to maximize a certain reward or minimize a given cost. This knowledge can either be built from scratch and is called model-free or uses a model of the environment to learn faster, which is called model-based. The agent environment is formulated as a Markov Decision Process (MDP) to be solved using RL algorithms. The model-free approach can be used to learn from the environment from scratch because it is hard to model the randomly changing service demands over time. RL is proven to excel in the areas of robotics, studying behaviors, scheduling, and many more [5]. Motivated by the advancement in RL, we propose in this paper Fscaler (Fog scaler), a model-free RL to solve our scaling problem taking into account the aforementioned challenges. Fscaler generates four combined decisions serving the horizontal resource scaling of containers in fog clusters based on defined cost functions. These four decisions are: (1) the number of containers to add, (2) the number of containers to remove, (3) an efficient placement of the service instances on fogs considering contradicting objectives and available resources, and (4) selection of fogs to stop and remove running instances that are not useful or can block other applications. Through a series of experiments, we obtain promising results showing the correctness and efficiency of our proposed MDP formulation after solving it using a model-free RL algorithm called SARSA. To the best of our knowledge, we are the first to propose the horizontal scaling of containers in fog clusters that is capable of making such decisions. The contributions of this paper are:

- A novel architecture to account for dynamic resource scaling of containers through integrating FScaler, an RL agent, in Kubernetes fog clusters.
- A novel MDP formulation of the scaling problem, which can be later solved by various RL agents depending on the fog cluster and application scale.
- Proposing the use of SARSA to build the FScaler agent, which is proven to reach optimal policy through a series of realistic experiments.

The rest of this paper is organized as follows. The related work is depicted in Section II. In Section III, we illustrate the integration of Fscaler in Kubernetes. In Section IV, we formulate the horizontal scaling and placement problems as MDP. In Section V, we present the SARSA algorithm to build FScaler. In Section VI, we present a series of experiments using real-world datasets of servers loads and capacities. We finally conclude with future directions in Section VII.

II. Related Work

In this section, we discuss different related proposals considering the problem of resource scaling in fog environments.

A. Resource Scaling and Placement in Fog Clusters

Authors in [6] studied the current trends and architectures of fog computing. In this survey, the authors highlighted the limitations of such architectures and pointed out the deployment issue of services in fog environments. The main problem is the ability to scale fog resources in order to achieve efficient placement of new services without affecting the running ones.

B. Resource Scaling Using RL

Many approaches used RL to horizontally and vertically scale resources on the cloud, taking into account the deployment cost [7]. This is different in the fog computing context, where containers are used for deployment and scaling, and placement happens under different conditions such as the change in resource availability of fogs and their different locations. Few works considered scaling containers in fog environment. For instance, Authors in [8] built an MDP for horizontally and vertically scaling containers resources inside one fog. This work considers scaling on multiple fogs having different locations. The limitation of [8] is the use of Linear Integer Programming to do the placement of newly created instances on fog devices, which might take time if the input is large and therefore delays the scaling procedure. On the other hand, FScaler is able to proactively do the scaling and placement at once using our Formulation.

C. Studying Behaviors and Scheduling Tasks Using RL

Several approaches have used RL to study users' behavior in terms of demands for services. One application where studying the demands is useful is caching. For instance, authors in [9] formulated their problem as MDP and used RL to decide on the files to cache in small base stations deployed in different locations based on the changes in users demands.

III. Proposed Architecture

The motivation behind our proposed scheme is to decide upon efficient scaling of containers resources in fog clusters in order to handle the demands of users and avoid overloading containers on fogs, which can result in its failure or instability. The challenges facing our approach can be summarized as the randomly changing demands of users, the randomly changing load of fogs and therefore the availability of resources, deciding on the exact number of containers to be added and/or removed at a time-step, and finally the proper placement and/or removal of containers on/from specific fogs.

In order to overcome these challenges, we propose the

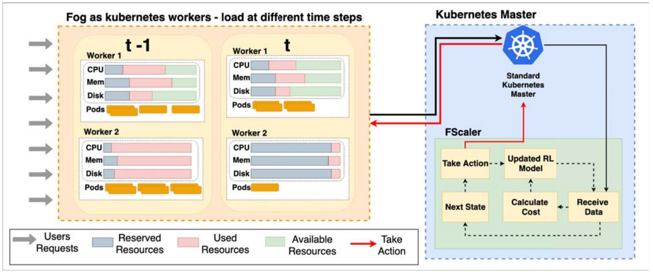


Fig. 1: Proposed Architecture

architecture illustrated in Fig. 1. The core of this architecture is FScaler, an RL agent that is capable of making the scaling and placement decisions. In short, fog devices receive requests from users. After serving these requests, the fog sends useful information about its performance and current resources state. Fscaler, in turn, utilizes this information to advance its knowledge and take effective actions for releasing and/or using more resources in its cluster. Fog and worker are used interchangeably throughout the paper with the same meaning.

Fogs as Kubernetes Workers: The cluster size represents the number of workers it has. For instance, two workers are considered in our sample case in Figure 1. Each device's resources containing CPU, Memory, and Disk are divided into three parts. First, resources reserved by other applications running on the fog (shown in gray). Second, resources consumed by containers running our application (shown in pink). The third is the available idle resources that can be used to add more instances for our application or can be used by other applications. Noting that containers in the case of Kubernetes clusters run in the form of pods. Resources utilized by pods of our application must not be idle for not wasting the worker resources. Besides, overloading pods can result in instability of the service and possibly affect its availability. Therefore, it is relevant to smartly use more of the workers' available resources in case needed, while not blocking other applications, to achieve load-balancing. It is also important to mention that pods addition or creation does not affect other instances running the service [4].

Kubernetes Master: Kubernetes master in our architecture is composed of two entities: The standard kubernetes master and the FScaler RL agent. These entities couple together to achieve the desired management and resource scaling of the cluster. The standard Kubernetes master role is to manage the state of the cluster by monitoring the performance of pods in every worker, restarting or creating new copies of failing ones, load balancing the incoming traffic on pods to avoid overloads, and scheduling the creation of new pods on available nodes. We disable the scheduling functionality of the traditional Kubernetes master and replace it with FScaler for placement and scheduling of the new instances based on defined cost functions that better serve the fog computing purpose, and study the changes in resources available on the workers. As shown in Figure 1, the Kubernetes master periodically receives data from its workers to be used by FScaler for learning. These data represent the response time of requests sent to the user, the current placement of containers, and the newly available resources of workers. FScaler is a model-free RL agent that periodically takes actions and learns by capturing data about its environment. FScaler interacts with the environment based on our MDP formulation detailed in the next section.

IV. Modeling FScaler

MDP is a mathematical framework for modeling sequential decision making in stochastic environments. An MDP model is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \gamma)$. $\mathcal{S}$ is a set of states that represents all possible states of the environment. $\mathcal{A}$ is the set of all possible actions or decisions to take. $\mathcal{P}$ is the transition probability matrix that represents a probability distribution over the next states s' after taking action a $\in \mathcal{A}$. Based on the Markov property, the agent transitions to a new state independent from all previous states and actions. $\mathcal{C}$ is the cost function evaluated after taking action a on state s and representing different objectives. $\gamma \in [0, 1]$ is the discount factor that decides on how much the agent cares about future rewards or costs for the current state. The ultimate goal is to obtain an optimal policy π^* that given state s, it outputs action $\pi^*(s)$ that minimizes the given cost. RL is a well-known technique used to find π.

In the context of our problem, we formulate the horizontal resource scaling of containers in orchestrated fog clusters as MDP, taking into consideration the change of users' demands and available resources of fogs over time. In our formulation, we consider scaling for one application. A single pod P of the application is to be scaled in the cluster, which requires resources described in terms of CPU, Memory, and Disk. A pod is represented as $P = [P_{CPU}, P_{Mem}, P_{Disk}]$. Replications or removal of pod instances can happen on any fog in the cluster. We denote by H the set of fogs in the cluster, where $H_i = [H_{i_{CPU}}, H_{i_{Mem}}, H_{i_{Disk}}]$. In our environment, the performance of fogs is measured by response time $U(t)$ of users' requests in ms at time t. Furthermore, we denote by k a constant in ms to represent the maximum response time users can tolerate. If $U(t) \geqslant k$, this means enough pods should be placed in the cluster to handle the excess of requests. In such a case, our model should intelligently decide on the number of pods to create in order to utilize the available resources in the cluster efficiently. On the other hand, if $U(t) < k$ no additional instances of pods should be created. For this reason, we denote $M(t)$ the placement vector of size $m \times 1$ of pods on m devices in the cluster. Supposing a Kubernetes cluster contains three workers ($m = 3$) and our agent decides to create five new pods on the cluster, a possible placement would be: $M(t) = (2, 2, 1)$. We also denote by $R(t)$ the vector of size $m \times 1$ the available resources of each fog in the cluster at time t, where each element of the vector contains information about the available CPU, Memory, and Disk. Available resources can change depending on the reserved resources by other applications running on the cluster. We assume that available resources on each node are normalized with respect to the total resources. A possible $R(t)$ can be $([0.8, 0.6, 0.4], [0.7, 0.5, 0.7], [0.9, 0.4, 0.5])$.

An action $a(t)$ in our action space $\mathcal{A}$ is a vector of size $m \times 1$, such that every entry $a_i(t) \in [-M_{max_i}, M_{max_i}] \mid i \in [1, m]$, and M_{max_i} is the maximum number of placements on worker i. M_{max_i} is calculated using (1):

$$M_{max_i} = min(\lfloor \frac{H_{i_{CPU}}}{P_{CPU}} \rfloor, \lfloor \frac{H_{i_{Mem}}}{P_{Mem}} \rfloor, \lfloor \frac{H_{i_{Disk}}}{P_{Disk}} \rfloor) \quad (1)$$

The action space is the set of all possible actions, taking into account that some actions will not be feasible based on the availability of resources for each worker/fog node. Therefore a single action combines four decisions discussed previously: number of new pods to create or remove, and the placement or removal of pods from worker nodes.

We denote by $s(t) \in \mathcal{S}$ a state in our state space, such that $s(t) = (U(t), M(t), R(t))$. Fscaler receives an update of the environment state periodically. Following Fig. 2, suppose that Fscaler receives information about state $s(t-1)$, Fscaler then takes a scaling and placement action $a(t)$ at time t to be performed on the environment. $M(t)$ is then calculated based on the previous placement. Workers then store information about the response time experienced by the user, as well as the resources available on each fog during the period t. For simplicity, $U(t)$ is then calculated based on the average of the responses, and $R(t)$ is calculated as the average of available resources over the period of t. The cost $\mathcal{C}_t$ is then calculated for state s_{t-1} and action a_t knowing U_t, M_t, and R_t. This cost can also be written as $\mathcal{C}_t((s_{t-1}, a_t | s_t))$. The next state s_t is then formed and forwarded to FScaler to take action a_{t+1}. This cycle repeats infinitely since our environment is continuous.

A. Cost function and placement objectives

The efficiency of our scaling decision in terms of the number of instances to add is measured by our response cost. The resource cost is also used to measure the amount of resources used by the scaled pods, which blocked other applications

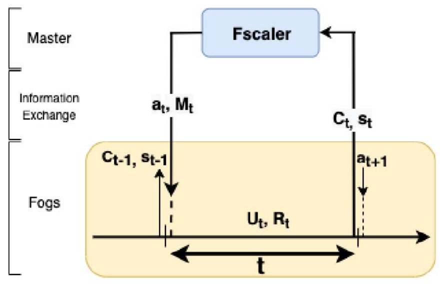


Fig. 2: The Steps to Get Our MDP Quantities Through Time

from running on the machine. This motivates the agent to intelligently manage the utilization of the resources on each fog in a way that allows other applications to run by learning the device's load over time. Last is our distance cost, which is dedicated to maintaining the proximity of fogs from requesting users in terms of distance. In other words, selecting fogs in the cluster that are furthest from the user is more expensive.

Our first cost $\mathcal{C}_{1,t}(s_t)$ is the response cost measuring the efficiency of the scaling decision at time t. This cost can be mathematically formulated as:

$$\mathcal{C}_{1,t}(s(t)) = \begin{cases} \frac{[U(t)-k]-[\epsilon\times\sum_{i=1}^{m} M_i(t)]}{U(t)}, & \text{if } 0 < \epsilon \times \sum_{i=1}^{m} \\ & M_i(t) < U(t) - k \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where ϵ is a constant decimal defined depending on the pod's performance towards minimizing the response time. Furthermore, $[\epsilon \times M(t)^T 1]$ denotes the response time minimized because of the pods scaling action a_t. Furthermore, we have two cases in (2). First is when our placement failed to improve the response time experienced by the user by $k - U(t)$. In this case, we take the response time that is experienced by the user after the placement using $[U(t) - k] - [\epsilon \times (M(t)^T 1)]$ and divide it by U_t for normalization. Otherwise, the cost is zero because the scaling was able to achieve the best in terms of response time.

The second cost $\mathcal{C}_{2,t}(s(t))$ is related to the resource consumption, which accounts for the amount of resources used by pods that blocked other applications from running on each fog during t. $\mathcal{C}_{2,t}(s(t))$ allows the agent to intelligently study the changes in resource usage of every fog during t where the placement of Fscaler already happened. To calculate this cost, we denote by $N(t)$ a vector of size $m \times 1$, representing the amount of additional resources needed for each fog to run other applications after the placement of Fscaler. Furthermore, $N_i(t)$ is a vector of size three containing the additional resources used for CPU, Memory, and Disk. $N_{i_{cpu}}(t)$ is calculated using (3). Similar calculations apply for $N_{i_{mem}}(t)$ and $N_{i_{disk}}(t)$:

$$N_{i_{cpu}}(t) = \begin{cases} (M_i(t) \times P_{cpu}) - R_i(t), & \text{if } M_i(t) \times P_{cpu} \\ & > R_{i_{cpu}}(t) \\ 0, & otherwise \end{cases} \tag{3}$$

Thereafter, $C_{2,t}(s(t))$ is computed as follows:

$$\mathcal{C}_{2,t}(s(t)) = \frac{\sum_{i=1}^{m} N_{i_{cpu}}(t) + N_{i_{mem}}(t) + N_{i_{disk}}(t)}{\sum_{i=1}^{m} Rmax_i} \tag{4}$$

In (4), R_{max_j} is the maximum resource available at fog j. The sum of R_{max_j} is used for normalization.

The third cost motivates the main purpose behind fog computing, which is bringing services closer to users. We measure this using a distance cost calculated as follows:

$$\mathcal{C}_{3,t}(s(t)) = \frac{\sum_{i=1}^{m} H_{i_D} \times M_i(t)}{\sum_{i=1}^{m} H_{max_D} \times M_i(t)} \tag{5}$$

In (5), H_D is a vector that contains the distances of each host in the cluster, assuming that the closest fog has $H_{i_D} = 0$. We take the final number of instances running in the cluster on each host and multiply it by the host distance. H_{max_D} is the distance furthest from the user and used for normalization.

Therefore, our cost function becomes:

$$C_t((s_{t-1}, a_t)|U_t, M_t, R_t) = \lambda_1 \times C_{1,t} + \lambda_2 \times C_{2,t} + \lambda_3 \times C_{3,t} \tag{6}$$

Where each $\lambda \in [0,1]$ is a weight multiplied by each cost function given $\sum_{i=1}^{3} \lambda_i = 1$. These weights are tuned depending on the requirement of the application and the nature of the cluster to give some cost functions more importance over the others, where the aim is to minimize C_t

B. Dynamics of the model

In $s(t)$, $U(t)$ and $R(t)$ are calculated after observing the behavior of users and usage of resources on fog nodes. Knowing that $U(t)$ and $R(t)$ are stochastic, we assume that the probability matrix $\mathcal{P}$ is unknown, which is practical. Therefore, Fscaler uses model-free RL algorithms to model the environment transitions through exploration and exploitation.

C. RL-based Solution

In RL, the agent tries to find the policy $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$ by minimizing a cost function, for instance (6). Action $a(t+1)$ in our MDP is incurred from the previous action and state, which tells the agent about the scaling and placement decisions to be performed. The performance of $a(t+1)$ is done using state value function $V_\pi(s)$, which is calculated by observing the rewards during an infinite time horizon. In order to do a model-free control to improve our policy, the agent should know $\mathcal{P}$ in order to figure out how to act greedily towards this value function. In our case, we do not know the dynamics of the environment. An alternative is to use the state-action value function $Q(s,a)$, where acting greedily is possible through maximizing or minimizing $Q(s,a)$ of a given state. Our aim in this paper is to find the optimal policy π^*, which can minimize C_t. The best π^* for FScaler is the one that allows the agent to make optimal scaling and placement decisions in its cluster. π^* can be expressed as follows:

$$\pi^*(s) = \arg\min_a Q^*(s,a) \ \ \forall s \in \mathcal{S} \tag{7}$$

In the next section, we present the Bellman equation used to optimize our policy π and introduce the use of SARSA for solving (7).

V. Solution for Small Scale Application

Exploration and exploitation are necessary for an agent to try different possibilities in the action space in order to properly act greedily. One of the ways to load balance exploration and exploitation is using ϵ-greedy policy improvement, which uses ϵ as a value that decays over time in order to reduce the number of explorations as the agent converges to an optimal policy π^*. In the context of our scaling problem, an RL agent evaluates his performance in a certain state and action by calculating a state-action value function $Q(s, a)$. In its simplest forms, RL agents can store $Q(s, a)$ in a matrix or QTable of size $|\mathcal{S}| \times |\mathcal{A}|$. This matrix is usually initialized to zero or to arbitrary values. SARSA is a model-free on-policy algorithm used to find the optimal Q-function Q^*. Q-values in SARSA are updated every time-step using the equation shown in (8) built using the Bellman equation. In (8), α is the learning rate, and s', a' are the next state and action.

$$Q(S, A) \leftarrow Q(S, A) + \alpha(C(S, A', S') + \gamma Q(S', A') - Q(S, A)) \tag{8}$$

In order to find the optimal policy based on our MDP, we use SARSA algorithm. The advantages of using SARSA are: (1) its ability to learn online from incomplete sequences; (2) its low variance to the true action value function and (3) it is proven to reach optimal policy [10]. The SARSA algorithm is shown in Algorithm 1.

Algorithm 1 FScaler Using SARSA

```
1:  **Initialize** $Q(s, a) = 0$, and $\forall s \in \mathcal{S}, a \in \mathcal{A}$ randomly
2:  **for** Each episode **do**
3:      Initialize $s$
4:      Select $a$ from $s$ using $\epsilon$-greedy
5:      **for** Each time-step of episode **do**
6:          Take action $a$, observe $\mathcal{C}, \mathcal{S}'$
7:          Select $a'$ from $s'$ using $\epsilon$-greedy
8:          Apply (8) to update $Q(s, a)$
9:          $s \leftarrow s'; a \leftarrow a'$;
10:     **end for**
11: **end for**
```

Despite the advantages of SARSA, SARSA for FScaler only works for small scale applications and clusters because of the "curse of dimensionality" problem.

VI. Numerical Tests

In this section, we experiment with FScaler using SARSA to show its efficiency in scaling and placing containers in fog clusters, while considering randomly changing users' demands and available resources. In order to prove that FScaler is able to converge to an optimal solution, we build two test cases with different input sizes, weights, and parameters. We then plot the cost function showing the performance of FScaler in each scenario. In order to prove that FScaler is indeed reaching optimal policy, we plot the response time the user is experiencing after the scaling and placement decisions, in addition to the amount of resources blocked on fogs because

Scenario	P_{CPU}	P_{Mem}	P_{Disk}	m	H_{1_D}	H_{2_D}	H_{3_D}	k	ϵ	Max Response
1	0.5	0.4	0.45	2	10	20	-	1	1	7
2	0.26	0.28	0.3	3	10	20	30	1	1	3

TABLE I: Experiments Settings

Scenario	γ	α	λ_1	λ_2	λ_3
1	0.9	0.1	0.4	0.4	0.2
2	0.9	0.1	0.3	0.5	0.2

TABLE II: SARSA Parameters and Cost Weights

of the scaling decision.
Two datasets are used to build our scenarios. The first one is WS-DREAM dataset [11], which maintains a set of QoS datasets collected from real web services. From this data, we built $U(t)$ from the response times in ms of the web services experienced by the user. The second dataset is Google cluster Trace 2011-2 [12], which illustrates a real set of available resources and demands of services that change over time. We built $R(t)$ from the normalized resources demands of three hosts from this dataset. In Table I, we show the experiments' settings of two scenarios. The choice of these parameters is based on various experiments to show the behavior of FScaler under different settings. As shown in Table I, the available CPU changes in both scenarios, as well as the number of hosts and their distances to study the performance and convergence of FScaler. In our experiments, we set a limit on the number of hosts and response time read from both datasets to avoid the curse of dimensionality problem. In Table II, we show the different parameters used to tune SARSA and the cost weights.

We ran Algorithm 1 multiple times for each scenario using the settings of Tables I and II. The results showing the costs evolution over time for each scenario are depicted in Fig. 3. As shown in this figure, FScaler is able to converge to a certain behavior after some time. Scenario 1 has a higher cost because m is small and P_{cpu} is large compared to scenario 2. However, FScaler in Scenario 1 converged faster compared to scenario 2 because the QTable is smaller in terms of states with fewer hosts. In order to prove that FScaler is reaching the optimal solutions in both scenarios, we display the results of Fig. 4 following scenario 2. In this scenario, $\lambda_1 < \lambda_2$, therefore giving more importance to minimize the resource consumption cost C_2 over the response cost C_1. As shown in the graph of Fig. 4a, the agent was able to minimize the average response to a value less than 1.6 ms. The best the agent can do is to

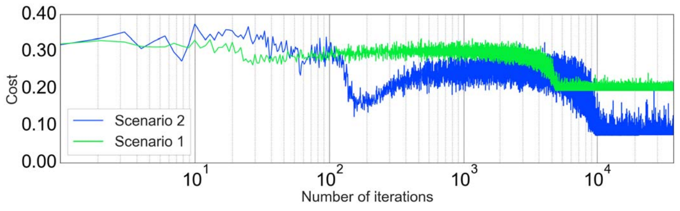
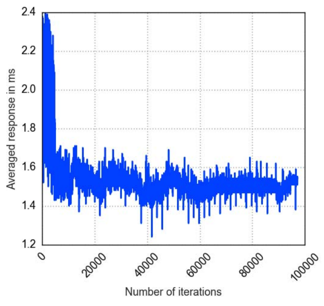
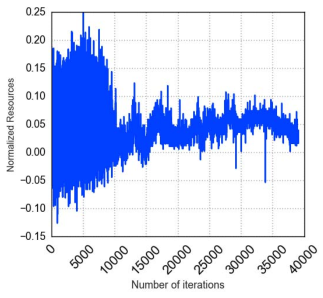


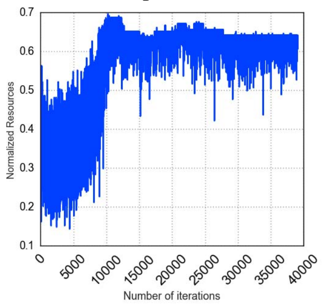
Fig. 3: FScaler Performance Using SARSA



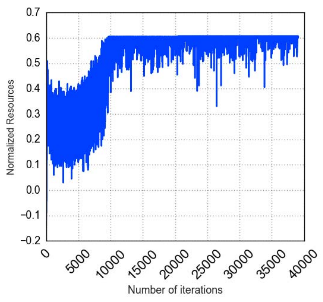
(a) Response Time



(b) Fog 1 - Available Resources



(c) Fog 2 - Available Resources



(d) Fog 3 - Available Resources

Fig. 4: Scenario 2: Response Time and Resources Available N Using FScaler

minimize this response to 0. However, because C_2 has higher importance, FScaler puts more attention to not block the new demands of fogs. Fig. 4b shows the resources available on fog one after the placement, where the agent utilizes all available resources without blocking other applications as the model converges. A negative value means the agent utilizes more than the available resources of the fog, which is not the case after convergence. Furthermore, the distance cost C_3 motivates the agent to use less resources on fogs far from the user. This is why the agent is not using much resources on fogs 2 and 3. This also explains why the agent did not drop the response time to 0 in Fig.4a. Therefore, the agent is able to behave optimally using SARSA in Scenario 2.

VII. Conclusion

In this paper, we propose FScaler, an RL agent capable of scaling and distributing containers based on the demands of users and available fog resources. We integrated FScaler in Kubernetes cluster architecture for easier management, scalability, and placement in containerized fog clusters. We were then able to model the scaling and placement problem as an MDP with three different costs: response, resources, and distance costs. Our model is able to accommodate the randomly changing users' demands and available resources of fogs in the cluster. Through our modeling, the agent is able to make the scaling and placement decisions at once. We then used SARSA in order to build FScaler that can intelligently manage the scaling and placement decisions by reaching the optimal policy. A series of experiments were conducted with real datasets that showed the ability of FScaler to converge by reaching the optimal policy. In our experiments, we considered two small test cases in terms of the cluster size and maximum possible response. In other scenarios, SARSA is not guaranteed to converge for larger test cases due to the curse of dimensionality problem. Therefore, our future direction is to build a function approximation for our formulation to form agents that can scale for large test cases. Finally, it is also important to note that this scaling approach can serve many other purposes such as distributing the load on the cloud to accommodate for unpredicted heavy demands like offloading tasks [13], improving security detections [14], and can be integrated into different clustering protocols [15], [16].

Acknowledgments

This work has been partially supported by the Lebanese American University, Concordia University, and Khalifa University.

References

[1] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big data and internet of things: A roadmap for smart environments*. Springer, 2014, pp. 169–186.

[2] H. Sami and A. Mourad, "Dynamic on-demand fog formation offering on-the-fly iot service deployment," *IEEE Transactions on Network and Service Management*, 2019.

[3] ——, "Towards dynamic on-demand fog computing formation based on containerization technology," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2018, pp. 960–965.

[4] G. Sayfan, *Mastering Kubernetes*. Packt Publishing Ltd, 2017.

[5] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis, "Reinforcement learning, fast and slow," *Trends in cognitive sciences*, 2019.

[6] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE access*, vol. 6, pp. 47 980–48 009, 2018.

[7] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[8] F. Rossi, "Self-management of containers deployment in decentralized environments," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642. IEEE, 2019, pp. 315–318.

[9] 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.

[10] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine learning*, vol. 38, no. 3, pp. 287–308, 2000.

[11] Z. Zheng and M. Lyu, "WS-Dream-Web Service QoS Datasets," *Retrieved from WS-Dream: http://www. wsdream. com/dataset. html*, 2012.

[12] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[13] H. Tout, C. Talhi, N. Kara, and A. Mourad, "Selective mobile cloud offloading to augment multi-persona performance and viability," *IEEE Transactions on Cloud Computing*, 2016.

[14] O. A. Wahab, J. Bentahar, H. Otrok, and A. Mourad, "Optimal load distribution for the detection of vm-based DDoS attacks in the cloud," *IEEE transactions on services computing*, 2017.

[15] H. Otrok, A. Mourad, J.-M. Robert, N. Moati, and H. Sanadiki, "A cluster-based model for QoS-OLSR protocol," in *2011 7th International Wireless Communications and Mobile Computing Conference*. IEEE, 2011, pp. 1099–1104.

[16] 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.