Published in the Proceedings of the 2019 IEEE International Conference on Cloud Computing (IEEE CLOUD 2019), pp. 329-338, Milan, Italy, July 8-13 2019. doi: 10.1109/CLOUD.2019.00061

Horizontal and Vertical Scaling of Container-based Applications using Reinforcement Learning

Fabiana Rossi, Matteo Nardelli, Valeria Cardellini
Dept. of Computer Science and Civil Engineering, University of Rome Tor Vergata, Italy
{f.rossi,nardelli,cardellini}@ing.uniroma2.it

***Abstract*—Software containers are changing the way distributed applications are executed and managed on cloud computing resources. Interestingly, containers offer the possibility of handling workload fluctuations by exploiting both horizontal and vertical elasticity "on the fly". However, most of the existing control policies consider horizontal and vertical scaling as two disjointed control knobs. In this paper, we propose Reinforcement Learning (RL) solutions for controlling the horizontal and vertical elasticity of container-based applications with the goal to increase the flexibility to cope with varying workloads. Although RL represents an interesting approach, it may suffer from a possible long learning phase, especially when nothing about the system is known a-priori. To speed up the learning process and identify better adaptation policies, we propose RL solutions that exploit different degrees of knowledge about the system dynamics (i.e., Q-learning, Dyna-Q, and Model-based). We integrate the proposed policies in Elastic Docker Swarm, our extension that introduces self-adaptation capabilities in the container orchestration tool Docker Swarm. We demonstrate the effectiveness and flexibility of model-based RL policies through simulations and prototype-based experiments.**

***Index Terms*—Reinforcement Learning, Elasticity, Self-adaptation, Container, Docker**

I. Introduction

The fast increasing adoption of container technologies calls for effective deployment and management strategies for containerized applications, also addressing their run-time adaptation [1]. Moreover, the ability of cloud computing to provide resources on demand encourages the development of elastic applications, which can be adapted in face of changing working conditions (e.g., variable workload). Containers allow to easily adapt the application deployment through horizontal and vertical scaling. Horizontal elasticity allows to increase (scale-out) and decrease (scale-in) the number of application instances (e.g., containers). Vertical elasticity allows to increase (scale-up) and decrease (scale-down) the amount of computing resources assigned to each application instance. Most of the existing solutions consider either horizontal (e.g., [2]) or vertical elasticity (e.g., [3], [4]). By fully exploiting elasticity, an application can more quickly react to small workload variations, through fine-grained vertical scaling, as well as to sudden workload peaks, through horizontal scaling. Nevertheless, so far only a limited number of works has explored the benefits of combining the two elasticity dimensions for container-based applications (e.g., [5]).

In this paper, we propose Reinforcement Learning (RL) techniques for adapting at run-time the deployment of container-based applications by means of horizontal and vertical elasticity. Differently from the popular threshold-based approaches used to drive elasticity (e.g., [3], [6]), we aim to design a flexible approach that can customize the adaptation policy without the need of manually tuning various configuration knobs. RL refers to a collection of trial-and-error methods by which an agent can learn to make good decisions through a sequence of interactions between the controlled system and the environment. As such, RL allows to express *what* the user aims to obtain, instead of *how* it should be obtained (as required by threshold-based policies). The adaptive nature of RL makes it very appealing to devise auto-scaling Cloud policies (e.g., [2], [7], [8]); within this context, RL approaches have been mostly applied to control horizontal elasticity of virtual machines (VMs). One of the main issues with RL policies is the possibly long learning phase, which is especially experienced when the algorithm assumes that nothing about the system dynamics is known a priori (*model-free* learning). An approach to boost the learning process is to provide the learner with basic knowledge about its environment (*model-based* learning). Therefore, together with the model-free Q-learning and the Dyna-Q algorithm [9], we propose a novel model-based RL approach that exploits what is known or can be estimated about the system dynamics to self-control the horizontal and vertical elasticity of container-based applications. We provide a general formulation that can take multiple deployment goals properly weighted into account (i.e., to minimize application performance penalty, adaptation cost, and resource cost) and be integrated in container orchestration tools. To show the benefits of our solution, we integrate it in Docker Swarm, thus realizing *Elastic Docker Swarm* (EDS), which extends Docker Swarm with self-adaptation capabilities. Resorting on a decentralized MAPE control loop [10], EDS can adapt the deployment of container-based applications at run-time in decentralized manner.

The main contributions of this paper are as follows. First, we design RL algorithms for controlling elasticity of container-based applications using a model-based RL approach. For sake of comparison, we also design approaches based on Q-learning and Dyna-Q algorithms (Sections III-IV). Second, we present a prototype implementation of the designed control policies in EDS (Section V), our extension of the well-known orchestration tool Docker Swarm. Third, we extensively evaluate the proposed solutions by means of simulations and prototype-based experiments (Section VI). In particular, we show the

flexibility and efficacy of using the proposed model-based RL solution with respect the other elasticity control policies.

II. Related Work

Cloud applications can be long-running and subject to varying workloads. Software containers enable to accordingly adapt the applications deployment at run-time, so to preserve their performance. We can classify the existing research works according to: (1) the scope, (2) the deployment goals, and (3) the actions and methodologies used to adapt the deployment.

To identify the *scope*, we observe that elasticity actions can be applied either at the *infrastructure level* [11] or at the *application level* [12]. At the infrastructure level, the elasticity controller changes the number of computing resources, usually by acquiring and releasing VMs, e.g., [2], [8], [13]. At the application level, the controller adjusts the computing resources directly assigned to the application (e.g., changing its parallelism degree [3], [12], [14]). A few solutions integrate the elasticity controller within the application code, i.e., embedded elasticity [11]; having no separation of concerns, the application itself should also implement mechanisms and policies steering the adaptation. Conversely, most research efforts use an *external controller* to carry out the adaptation actions (e.g., [2], [6]); this approach improves software modularity and flexibility. To exploit such benefits, we propose an external controller to manage horizontal and vertical elasticity of distributed containerized applications.

The elasticity of containers (and VMs) is carried out in order to achieve different objectives: to improve application performance (e.g., [6]), load balancing and resource utilization (e.g., [15], [16]), energy efficiency (e.g., [17]), and to reduce the deployment cost (e.g., [3], [5], [12]). Few works (e.g., [18]) consider a combination of deployment goals, as we also do.

The *actions* that control the deployment of container-based applications include the placement of containers on the underlying computing nodes, their horizontal or vertical scaling, and their migration (e.g., [19]). Elliott et al. [19] present a novel approach to container management that enables the rapid live migration of stateful containers between hosts belonging to different cloud infrastructures. When containers are placed on VMs, a second level of deployment can entail the VM allocation onto the physical computing resources. Most of the works consider a single level of deployment (e.g., [3], [12], [18]), while few works solve a multi-level problem [5], [6]. In this paper, we consider only a single level of deployment and exploit elasticity at the containers level.

To determine or adapt the deployment of container-based applications, the existing approaches recur to two main *methodologies*: mathematical programming and heuristics. Mathematical programming approaches consider the initial placement of containers (e.g., [16], [18]) as well as their run-time deployment adaptation (e.g., [5], [12], [15]). Differently from this paper and [5], the above works do not take into account the adaptation cost, i.e., the performance penalty resulting by the deployment reconfiguration. We can classify the mostly used heuristics in: custom solutions (e.g., [16], [18]), threshold-based (e.g., [3], [6]) and RL-based solutions. Threshold-based policies represent the most popular approach to scale containers at run-time, as well as for the cloud infrastructure layer. Orchestration frameworks that support container scaling (e.g., Kubernetes, Docker Swarm, Amazon ECS) usually rely on best-effort threshold-based policies based on some load metrics (e.g., CPU utilization). However, to be effective threshold-based policies need to properly set the threshold parameters, which may require some knowledge of the application resource consumption. As such, setting these thresholds can result in a cumbersome task. To identify suitable thresholds, Barna et al. [6] estimate performance metrics exploiting a layered queuing network model of the system. Vertical elasticity solutions for cloud applications have been explored only in few works (e.g., [3], [4]). ElasticDocker [3] employs a threshold-based policy to vertically scale the container resources, while Shekhar et al. [4] propose a data-driven and predictive framework based on machine learning techniques in order to build a runtime model of the system performance. Nevertheless, they do not consider the combination of vertical and horizontal scaling.

RL represents an interesting approach for the run-time self-management of cloud systems, where it has been mostly applied to devise policies for VM allocation and provisioning (e.g., [7], [8], [11]) and in more limited way to manage containers (e.g., [2]). Arabnejad et al. [7] combine the Q-learning and SARSA RL algorithms with a fuzzy inference system that drives VM auto-scaling. Horovitz et al. [2] propose a threshold-based policy for horizontal container elasticity that uses Q-learning to adapt the scaling thresholds. Most works have considered the classic Q-learning and SARSA algorithms [9]; however, being model-free solution, they suffer from slow convergence. To tackle this issue, Tesauro et al. [8] propose a hybrid RL method to dynamically allocate homogeneous servers to multiple applications. In the different field of distributed data stream processing, a model-based RL approach is presented in [20], which however only controls the horizontal elasticity of the application operators. Function approximation represents another, orthogonal approach to solve the slow convergence rate of RL: by approximating the system state or the action-value function, the agent can explore a reduced number of system configurations before learning a good adaptation policy [9]. Tang et al. [21] propose a RL-based solution for migrating containers deployed on fog computing nodes. Interestingly, to deal with the large number of system states, the authors integrate a deep neural network within the Q-learning algorithm. Differently from these works, we propose a novel model-based RL policy that exploits system knowledge, so to reduce the learning phase and improve the adaptation policy quality. Moreover, we investigate the benefits of jointly exploiting vertical and horizontal elasticity to adapt at run-time the application deployment.

III. System Model and Problem Definition

We consider a very general application model, where the application is a black-box entity that carries out specific tasks

(e.g., perform computation, access data sets). To properly process increasing incoming workloads, multiple application instances can be created and executed in parallel. Each instance works autonomously and processes a subset of the incoming requests. The application exposes requirements on its response time, expressed in terms of a target response time that should not be exceeded (i.e., $R_{\max}$). To simplify the application deployment and run-time management, software containers can be used (e.g., Docker).

At run-time, the application can be subject to varying workloads. To meet its performance requirement, the amount of computing resources granted to the application should be dynamically changed in an efficient way. To this end, we can exploit both vertical and horizontal elasticity, which, nonetheless, can introduce performance penalties (e.g., downtime), caused by the enactment of the adaptation actions. The elasticity of the container-based application should be properly driven so to guarantee the application performance, while minimizing resource wastage, and adaptation costs.

IV. Horizontal and Vertical Elasticity with RL

RL approaches aim to learn the optimal adaptation strategy through direct interaction with the system [9]. RL strategies aim to learn what to do (i.e., how to map situations to actions), so to minimize a numerical cost signal. One of the challenges that arises in RL is the trade-off between *exploration* and *exploitation*. To minimize the obtained cost, a RL agent must prefer actions, tried in the past, that it found to be effective (exploitation). However, to discover such actions, it has to explore new actions (exploration).

For each application, we consider a RL agent that is in charge of adapting at run-time the application deployment with the aim of minimizing a long-term cost.

The RL agent interacts with the application in discrete time steps. At each time step, the agent observes the application state and performs an action. One time step later, the application transits in a new state, causing the payment of an immediate cost. Both the paid cost and the next state transition usually depend on external unknown factors. To minimize the expected long-term cost, the agent estimates the so-called Q-function. It consists in $Q(s,a)$ terms, which represent the expected long-term cost that follows the execution of action a in state s. The Q-function is used to take scaling decisions: given the system state s, the agent performs the action a that minimizes $Q(s,a)$. By observing the actual incurred costs, $Q(s,a)$ is updated over time, thus improving the scaling policy. The different RL approaches adopt distinct strategies for estimating the expected long-run cost.

We define the application state at time i as $s_i = (k_i, u_i, c_i)$, where k_i is the number of containers (i.e., application instances), u_i is the CPU utilization, and c_i is the CPU share granted to each container. We denote by $\mathcal{S}$ the set of all the application states. Even though the CPU utilization (u_i) and CPU share (c_i) are real number, we discretize them by assuming that $u_i \in \{0, \bar{u}, ..., L\bar{u}\}$ and $c_i \in \{\bar{c}, 2\bar{c}, ..., M\bar{c}\}$, where $\bar{u}$ and $\bar{c}$ are suitable quanta. We also assume that the number of containers k ranges in the interval $\{1, 2, ..., K_{\max}\}$, where $K_{\max}$ is the maximum application replication degree.

For each state $s \in \mathcal{S}$, we have a set of *feasible* adaptation actions $\mathcal{A}(s) \subseteq \mathcal{A}$, where $\mathcal{A}$ is the set of all actions. We propose two different action models: the **5-action** model and the **9-action** model. In the 5-action model, we can either perform horizontal or vertical scaling, while in the 9-action model we can *jointly* perform the two dimensions of scaling. Formally, the 5-action model consists of $\mathcal{A} = \{-r, -1, 0, 1, r\}$, where $\pm r$ represents a vertical scaling (i.e., $+r$ to add CPU share and $-r$ to remove CPU share), ± 1 represents a horizontal scaling (i.e., $+1$ to scale-out and -1 to scale-in), and $a = 0$ is the *do nothing* decision. Alternatively, the 9-action model consists of $\mathcal{A} = \{-1, 0, +1\} \times \{-r, 0, r\}$. Obviously, not all the actions are available in any application state: e.g., in 5-action model, if the state s has $k = K_{\max}$ and $c = M\bar{c}$, the available actions are $\mathcal{A}(s) = \{-r, -1, 0\}$ (i.e., we cannot perform further scale up and out operations).

To each triple (s, a, s') we associate an immediate cost function $c(s, a, s')$, which captures the cost of carrying out action a when the application state transits from s to s'. The RL agent wants to minimize the cost so to (i) reduce the number of adaptations, (ii) keep satisfying application performance, and (iii) limit resource wastage. For this purpose, the cost function includes three different contributions:

- the adaptation cost c_{adp}, which accounts for the application unavailability following an adaptation. For the purpose of learning the adaptation policy, it suffices to consider a simplified model that introduces a constant penalty for vertical scaling actions. Indeed, horizontal scaling decisions do not compromise the application availability, because they can be performed by simply starting/terminating containers;
- the performance penalty c_{perf}, paid whenever the application exceeds the response time bound $R_{\max}$;
- the resource cost c_{res} for running the application. We assume that the cost is proportional to the number of application instances and assigned CPU share.

We combine the different costs into a single weighted cost function, where the different weights allow us to express the relative importance of each cost term. Formally, we define the immediate cost function $c(s, a, s')$ as the weighted sum of the costs, normalized in the interval $[0, 1]$:

$$\begin{aligned} c(s,a,s') &= w_{\text{adp}} \frac{\mathbb{1}_{\{vertical-scaling\}} c_{\text{adp}}}{c_{\text{adp}}} + \\ &\quad + w_{\text{perf}} \frac{\mathbb{1}_{\{R(k+a_1, u', c+a_2) > R_{\max}\}} c_{\text{perf}}}{c_{\text{perf}}} + \\ &\quad + w_{\text{res}} \frac{(k+a_1)(c+a_2) c_{\text{res}}}{K_{\max} \cdot c_{\text{res}}} \\ &= w_{\text{adp}} \mathbb{1}_{\{vertical-scaling\}} + \\ &\quad + w_{\text{perf}} \mathbb{1}_{\{R(k+a_1, u', c+a_2) > R_{\max}\}} + \\ &\quad + w_{\text{res}} \frac{(k+a_1)(c+a_2)}{K_{\max}} \end{aligned} \tag{1}$$

Algorithm 1 Dyna-Q

```
1: Initialize Q(s,a) and Model(s,a), ∀s ∈ S, ∀a ∈ A(s)
2: while true do
3:     s ← observe the application state
4:     a ← select an action using the current estimate of Q
5:     Observe the next state s' and the incurred cost c
6:     Update Q(s,a) using Eq. 2
7:     Model(s,a) ← c, s'
8:     for i = 0 → n do
9:         s ← random state previously observed
10:        a ← random action previously taken in s
11:        c, s' ← Model(s,a)
12:        Update Q(s,a) using Eq. 2
13:    end for
14: end while
```

where $\mathbb{1}_{\{\cdot\}}$ is the indicator function, w_{adp}, w_{perf} and w_{res}, $w_{\text{adp}} + w_{\text{perf}} + w_{\text{res}} = 1$, are non negative weights for the different costs, and $R(k,u,c)$ is the application response time in the state $s = (k,u,c)$. Furthermore, we decompose action a in terms of number of containers added/removed, a_1, and amount of CPU share increased/decreased, a_2.

We consider three different RL approaches that differ for the actual learning algorithm adopted and on the assumptions about the system. We first consider the simple Q-learning algorithm; it is a *model-free* algorithm that requires no knowledge of the system dynamics. Then, we present Dyna-Q, which builds a system model on the basis of the real experience. Furthermore, we propose a *model-based* approach, which exploits the known (or estimate) system dynamics to accordingly update the Q-function. The model-based solution enriches RL agents with a model of the system, thus driving the exploration actions to speed up the learning phase.

A. Q-learning

Q-learning essentially estimates the optimal Q-function, Q^*, by its sample averages [9]. In this paper, we consider the simple ϵ-greedy action selection method: at any decision step i, with probability ϵ, Q-learning chooses a random action to improve its knowledge of the application, whereas, with probability $1-\epsilon$, it chooses the greedy action by exploiting its knowledge about the application (i.e., $a_i = \arg\min_{a \in A(s_i)} Q(s_i,a)$). Most of the time the ϵ-greedy policy selects the best known action for a particular state, while it favors the exploration of sub-optimal actions with low probability. At the end of each time slot i, $Q(s_i,a_i)$ is updated as follows:

$$Q(s_i,a_i) \leftarrow (1-\alpha)Q(s_i,a_i) + \alpha\left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1},a')\right] \tag{2}$$

where $\alpha \in [0,1]$ is the *learning rate* parameter and $\gamma \in [0,1)$ is the *discount factor*. Observe that (2) simply updates the old estimate of Q with the just observed values, such as the observed cost c_i and the discounted cost expected when the system is in s_{i+1}, that is $\min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1},a')$.

B. Dyna-Q

Differently from Q-learning, Dyna-Q aims to speed up the learning process by simulating the system interaction with the environment [9]. Algorithm 1 summarizes the Dyna-Q learning. At run-time, Dyna-Q observes the application state and selects an adaptation action using the estimates of $Q(s,a)$, as Q-learning does. At the end of the time step i, Dyna-Q exploits a sampled model of the system, $Model(s,a)$, to simulate the interaction between the application and the environment (lines 8–13). Dyna-Q updates $Model(s,a)$ at run-time, by storing the next state s' and cost c for the explored state-action pair (s,a), see line 7. Assuming a deterministic environment, Dyna-Q updates the Q-function using (2) and resorting on the state-action pairs previously observed.

C. Model-Based Reinforcement Learning

As third strategy, we consider a *full backup model-based* RL approach (see [9]). In the full backup approach, we rely on a possibly approximated system model and directly use the Bellman equation to compute the Q-function:

$$Q(s,a) = \sum_{s' \in \mathcal{S}} p(s'|s,a)\left[c(s,a,s') + \gamma \min_{a' \in \mathcal{A}(s')} Q(s',a')\right] \quad {\scriptstyle \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)} \tag{3}$$

We replace the unknown transition probabilities $p(s'|s,a)$ and the unknown cost function $c(s,a,s')$, $\forall s,s' \in \mathcal{S}$ and $a \in A(s)$, by their empirical estimates.

To estimate $p(s'|s,a)$, it is sufficient to estimate the CPU utilization transition probabilities $P[u_{i+1} = u' | u_i = u]$. In fact, we observe that:

$$\begin{aligned} p(s'|s,a) &= P[s_{i+1} = (k',u',c') | s_i = (k,u,c), a_i = a] \\ &= \begin{cases} P[u_{i+1} = u' | u_i = u] & k' = k + a_1 \wedge \\ & \wedge\, c' = c + a_2 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{4}$$

where $a = (a_1,a_2)$ is the scaling action, defined in term of the updated number of containers (a_1) and updated amount of CPU share (a_2). Since u takes value in a discrete set, we will write $P_{j,j'} = P[u_{i+1} = j'\bar{u} | u_i = j\bar{u}]$, $j,j' \in \{0,\ldots,L\}$ for short. Let $n_{i,jj'}$ be the number of times the CPU utilization changes from state $j\bar{u}$ to $j'\bar{u}$, in the interval $\{1,\ldots,i\}$, $j,j' \in \{0,\ldots,L\}$. At time i, the transition probability estimates are $\widehat{P_{j,j'}} = n_{i,jj'} / \sum_{l=0}^{L} n_{i,jl}$, and, via (4), we estimate $\hat{p}(s'|s,a)$.

For the estimates of the immediate cost $c(s,a,s')$, we observe that it can be written as the sum of two terms, respectively named as the known and the unknown cost:

$$c(s,a,s') = c_k(s,a) + c_u(s') \tag{5}$$

The *known cost* $c_k(s,a)$ depends on the current state and action; in our case, it accounts for the adaptation and resource costs. The *unknown cost* $c_u(s')$ depends on the next state s'. As in (1), $c_u(s')$ accounts for the performance penalty. As we assume that the application model is not known, we have

Algorithm 2 Model-Based Reinforcement Learning Update

1: Update estimates $\widehat{P_{j,j'}}$ and $\hat{c}_{u,i}(s_i)$
2: **for all** $s \in \mathcal{S}$ **do**
3: **for all** $a \in \mathcal{A}(s)$ **do**
4: $Q(s,a) \leftarrow \sum_{s' \in \mathcal{S}} \hat{p}(s'|s,a) \cdot$
5: $\cdot \left[\hat{c}(s,a,s') + \gamma \min_{a' \in \mathcal{A}(s')} Q(s',a')\right]$
6: **end for**
7: **end for**

to estimate $c_u(s')$ online. Therefore, at time i, the RL agent observes the immediate cost c_i and estimates $c_{u,i}(s')$ as:

$$c_{u,i}(s') = c_i - c_{k,i}(s,a)$$

We use the sample value $c_{u,i}(s')$ to update the estimate of the unknown cost $\hat{c}_{u,i}(s')$, as follows:

$$\hat{c}_{u,i}(s') \leftarrow (1-\alpha)\hat{c}_{u,i-1}(s') + \alpha c_{u,i}(s') \tag{6}$$

The estimate of the unknown cost $\hat{c}_{u,i}(s')$ is then used to compute the cost of applying a in s according to (5). Given a state $s = (k,u,c)$, we can heuristically assume that in the next state $s' = (k',u',c')$ the expected cost due to $R_{\max}$ violation is not lower when the number of containers is reduced, the CPU utilization increases, and/or the CPU share is reduced. Vice versa is also true. Therefore, while updating $\hat{c}_{u,i}(s)$, $\forall s \in \mathcal{S}$, we can enforce the following properties:

$$\hat{c}_{u,i}(s) \leq \hat{c}_{u,i}(s') \qquad \forall k \geq k', u \leq u' c \geq c'$$
$$\hat{c}_{u,i}(s) \geq \hat{c}_{u,i}(s') \qquad \forall k \leq k', u \geq u', c \leq c'$$

Algorithm 2 summarizes the model-based RL update steps.

V. Docker-based Implementation

A. Docker Swarm

Docker is an open-source platform to create, deploy, and manage containerized applications. A Docker container is an instance of a container snapshot (or image), which contains the application together with all the data needed for its execution (e.g., dependencies, configuration file). Docker comes with a Docker Engine that allows to build and run containers, using REST APIs or a command-line interface. Docker allows to configure a container with specific resource quota, which limits the amount of (CPU and memory) resources the container can use on the hosting machine. At run-time, the resource quota can be updated, thus realizing vertical elasticity. To easily allocate multiple containers on distributed computing resources, Docker integrated the *swarm mode* with the Docker Engine from version 1.12 [22]. It enables to cluster Docker-enabled nodes and simplify the execution of containers across multiple nodes. The Docker Swarm architecture follows the master-workers pattern. The master deals with the orchestration and scheduling of containers. It also manages the swarm by accepting other nodes as workers. A *worker* provides its computational capability to the swarm, enabling the distributed execution of containers. Docker Swarm uses its scheduling capabilities to allocate containers to the workers. The default scheduling strategy is *spread*, which distributes containers so to optimize for the node with the least number of containers.

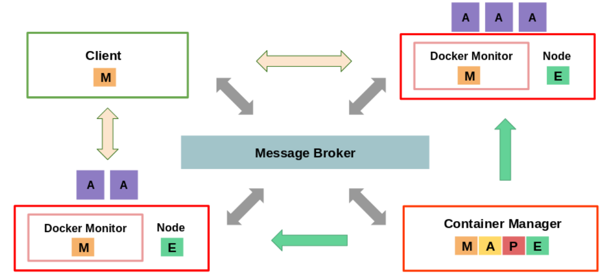


Fig. 1: Architecture of Elastic Docker Swarm

B. EDS: Elastic Docker Swarm

To enable autonomic capabilities in Docker, we propose Elastic Docker Swarm (for short, EDS). It extends the Docker Swarm architecture so to introduce the MAPE control loop [23]. The latter includes four main components (Monitor, Analyze, Plan and Execute) that are responsible for the self-adaptation functions. The Monitor collects data about the application and the execution environment. The Analyze component uses the collected data to determine whether an adaptation is beneficial. If the adaptation is needed, the Plan component determines an adaptation plan for the application, which is enacted through the Execute component. The modularity and the broad presence of APIs allowed us to easily integrate our MAPE components in Docker. Being loosely coupled with the Docker Swarm architecture, our components are general enough and could be integrated with other orchestration tools. We architect EDS following the *master-workers pattern*, used to decentralize the MAPE control loop [10]. In particular, it includes a single master component, which runs the Analyze and Plan phases, and multiple independent worker components, which run the Monitor and Execute phases in a decentralized manner. Having a single and centralized Analyze and Plan component, the master-workers pattern can be easily equipped with self-adaptation policies that determine when and how an application reconfiguration should be performed. Fig. 1 represents the architecture of EDS. The decentralized Docker Monitors and the Client realize the MAPE Monitor component. A *Docker Monitor* runs on each node of the swarm; it periodically publishes, on the message broker Apache Kafka, information about CPU utilization of containers running on the node. The *Client* collects, and periodically publishes on the message broker, the application response time. The *Container Manager* is the centralized control entity. First, it receives the monitoring information through the message broker. Then, it uses the RL agent to perform the Analyze and Plan phases. Specifically, in the Analyze phase, the RL agent determines the application state and updates the Q-function (see Section IV). In the Plan phase, the manager uses the RL agent to identify the scaling actions to be performed. To adapt the application deployment, EDS runs the Execute phase, which, in its turn, leverages on the Docker Swarm APIs. A vertical scale changes the container configuration: this causes the unavailability of the entire application for the time needed to enact the new config-

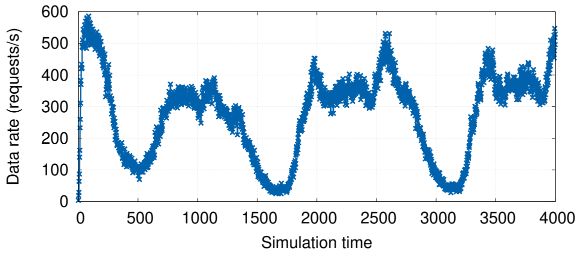


Fig. 2: Application workload used in simulation.

uration. A horizontal scale changes the number of containers used to run the application. Although this operation should introduce practically no downtime, performing a scale-out in Docker Swarm introduces an adaptation cost. This depends on the traffic routing strategy used by Docker Swarm, which directs the application requests to the newly added instance, even if it is not yet running. This introduces an adaptation cost inversely proportional to the number of application instances.

VI. Experimental Results

We thoroughly evaluate the proposed RL approaches using simulation and EDS-based experiments.

A. *Simulation Results*

By means of simulations, we evaluate the proposed RL policies with a threefold objective. First, we compare the model-based RL approaches against the model-free approach. Second, we investigate the benefits of using a 5-action or a 9-action model. Third, we show the flexibility of RL approaches that, with different cost function weights (Eq. 1), can learn different adaptation policies, so to accordingly avoid $R_{\max}$ violations, resource wastage, or frequent adaptations.

We consider a reference application modeled as an M/D/k_i queue, because we can reasonably assume that: the application receives random and independent (M) requests, its service time is deterministic (D), and the number of servers is equal to the number of containers (k_i) used at the time step i. We set the target response time $R_{\max} = 50$ ms and the service rate $\mu = 200 \cdot c_i$ requests/s, where $c_i \in (0, 1]$ is the amount of CPU share. Moreover, we consider that the application receives a number of requests that changes over time according to the workload pattern shown in Fig. 2. This workload pattern results by replaying the real-application data set, collected by Chris Wrong [24], accelerated by a factor of 1800 (i.e., 30 minutes of events are replayed in 1 second).

The RL algorithms use the following parameters: discount factor $\gamma = 0.99$ and, for Q-learning and Dyna-Q, learning rate $\alpha = 0.1$ and $\epsilon = 1/i$, where i is the simulation time. To discretize the application state, we use $\bar{u} = 0.1$ and $\bar{c} = 10\%$. We run the simulation on a machine with an Intel Core i7-4700MQ (8 cores at 2.40 GHz) and 8 GB of RAM. Tables I and II report the simulation results for the 5-action and 9-action model, respectively.

When we consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$ and $w_{\text{adp}} = 0.01$, avoiding $R_{\max}$ violations is very important. By comparing Tables I and II, we observe that, in general, the 9-action model slows down the learning process, because the agent should learn the impact of performing two actions at once (e.g., scale-up and scale-out). Q-learning and Dyna-Q obtain a similar number of $R_{\max}$ violations (around 25–27%). Conversely, the model-based solution learns a better elasticity policy that successfully controls the application with only 2.85% of response time violations. From Table I, we can see that the 5-action model simplifies the learning task, and all the RL policies reduce the number of $R_{\max}$ violations. With Q-learning, the application has a response time that exceeds $R_{\max}$ for 18% of the time. Dyna-Q reduces this value to 7%. Exploiting the system knowledge, the model-based solution further reduces the violations to 2%.

We now consider the case when saving resources is more important than the other objectives, i.e., $w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, and $w_{\text{adp}} = 0.01$. Intuitively, the agent should learn how to improve resource utilization at the expense of a high application response time (i.e., that exceeds $R_{\max}$). We can observe that the model-based solution successfully does it when both the 5-action and the 9-action models are used (see Tables I and II). The model-based solution runs the application with 1.09 instances, on average, that can access only to 11% of CPU resources; this represents the lowest amount of resources assignable to the application. Run-time adaptations are also avoided. As a consequence, the application is overloaded and the resulting median response time is unbounded. Regardless of the adopted action model, Q-learning and Dyna-Q struggle to find a stable configuration (77–94% of adaptations). Furthermore, although they lead to an average CPU utilization of about 80%, they employ a higher number of containers that can access to more resources (on average, 3 containers allocated to 50% of computing resources). In this case, it is not easy to capture the benefits of using 5 or 9 actions.

As third case, we balance the importance of three deployment goals, i.e., $w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$. Table I shows that the 5-action model allows to reduce the number of $R_{\max}$ violations and adaptations compared to 9-action model (Table II). Q-learning and Dyna-Q have 19% of $R_{\max}$ violations (instead of 36–40% with 9 actions), at the expense of under-utilizing computing resources (54–56% average resource utilization, instead of 65–69% with 9 actions). Also in this case, the model-based solution learns a better adaptation strategy, which only slightly depends on the number of actions. With 5 actions, the model-based strategy reduces $R_{\max}$ violations to 17%, with a 69% of average resource utilization. On average, it runs the application with 2.5 containers, each of which can use 86% of the assigned CPU. To achieve such trade-off, this RL strategy decides to adapt the application for almost 45% of the time. From Table I, we can readily observe that such behavior is needed to meet $R_{\max}$ requirements in face of changing workload.

Discussion. This set of experiments has shown the importance of providing system knowledge to improve the learning task. As such, the model-based solution uses the experience to estimate only the unknown system dynamics. Interestingly,

TABLE I: Simulation-based analysis: Application performance under different configurations of cost function weights and RL policies, when the 5-action adaptation model is used.

Weights	*Policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median* R (ms)	*Adaptations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**Q-learning**	17.87	55.83	62.84	4.49	13.57	76.06
	Dyna-Q	7.12	48.66	80.21	3.88	8.97	87.98
	Model-based	**2.37**	60.54	87.62	2.53	10.39	39.67
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**Q-learning**	46.16	72.63	54.34	3.49	35.66	77.38
	Dyna-Q	56.64	79.83	53.79	2.95	$+\infty$	91.45
	Model-based	99.80	**99.85**	**11.01**	**1.09**	$+\infty$	3.25
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**Q-learning**	19.32	55.89	68.50	4.28	11.60	71.26
	Dyna-Q	19.52	53.68	73.23	3.95	9.41	84.48
	Model-based	**17.17**	**69.01**	86.12	2.48	12.04	**45.06**

TABLE II: Simulation-based analysis: Application performance under different configurations of cost function weights and RL policies, when the 9-action adaptation model is used.

Weights	*Policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median* R (ms)	*Adaptations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**Q-learning**	27.17	62.82	61.32	3.87	15.59	88.98
	Dyna-Q	25.77	63.39	62.60	3.56	15.29	92.53
	Model-based	**2.85**	60.73	87.43	2.57	10.15	37.32
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**Q-learning**	61.18	80.95	46.62	3.08	$+\infty$	89.38
	Dyna-Q	62.58	81.89	46.32	3.70	$+\infty$	94.30
	Model-based	99.80	**99.85**	**11.04**	**1.09**	$+\infty$	2.95
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**Q-learning**	39.69	69.22	53.62	3.89	25.00	85.70
	Dyna-Q	35.64	65.08	54.61	4.35	20.53	91.10
	Model-based	**19.50**	**70.75**	78.35	2.56	15.16	**40.29**

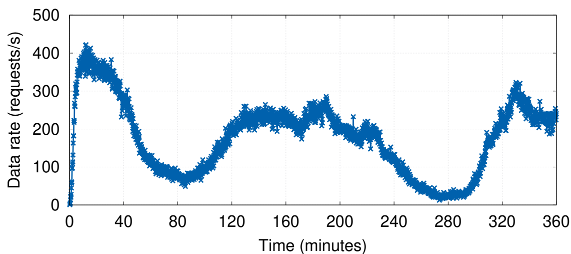


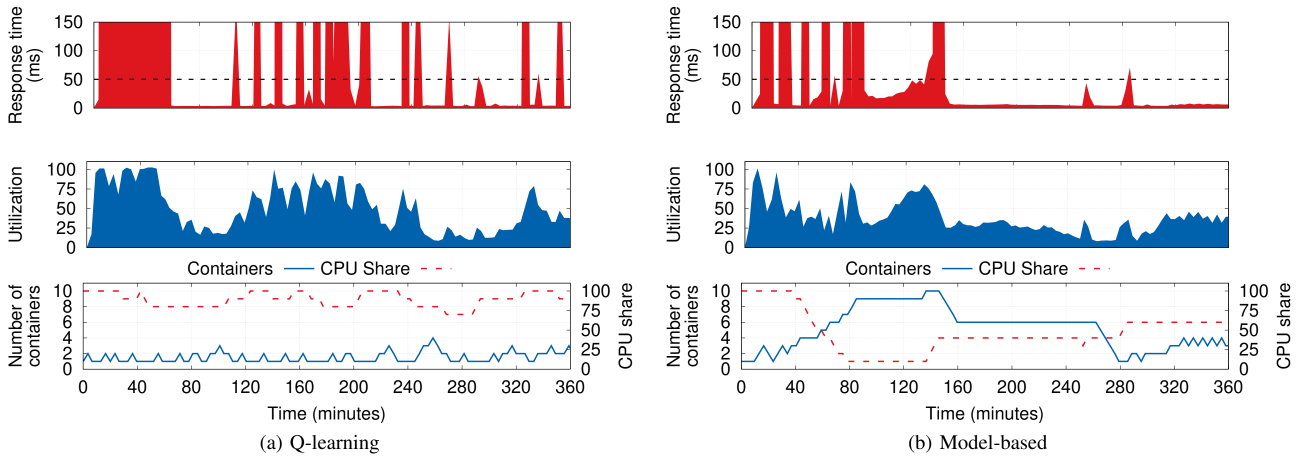
Fig. 3: Workload used in the prototype-based experiments.

it obtains best performance in every considered scenario. The experiments also show that the 9-action model increases the combination of state-actions to be evaluated, slowing down the learning task. Nevertheless, the model-based policy successfully exploits the 9-action model to jointly perform vertical and horizontal scaling and reduce the number of adaptations.

B. Prototype-based Results

Encouraged by the positive simulation results, we evaluate the RL algorithms in a real environment. To this end, we integrate the proposed policies in EDS. In particular, we compare the model-based RL solution against Q-learning, when both the 5-action and the 9-action models are considered. Due to space limitation, we do not consider Dyna-Q, which nonetheless introduces only a limited improvement over Q-learning. We deploy EDS on a cluster of 4 Amazon EC2 instances with 2 vCPUs and 8 GB of RAM (i.e., t2.large). The reference application computes upon request the sum of the first n elements of the Fibonacci sequence (complexity $O(n^2)$). As shown in Fig. 3, the application receives a varying number of requests. In this case, the incoming workload follows the load pattern resulting by replaying the reference data set [24] so to stress the resource usage by the application. The application requires $R_{\max} = 50$ ms as target response time. EDS executes the Analyze phase of the MAPE control loop every 3 minutes. Tables III and IV report the experimental results for the 5-action and the 9-action models, respectively; for sake of comparison, we report in Table V the application performance resulting by a static deployment.

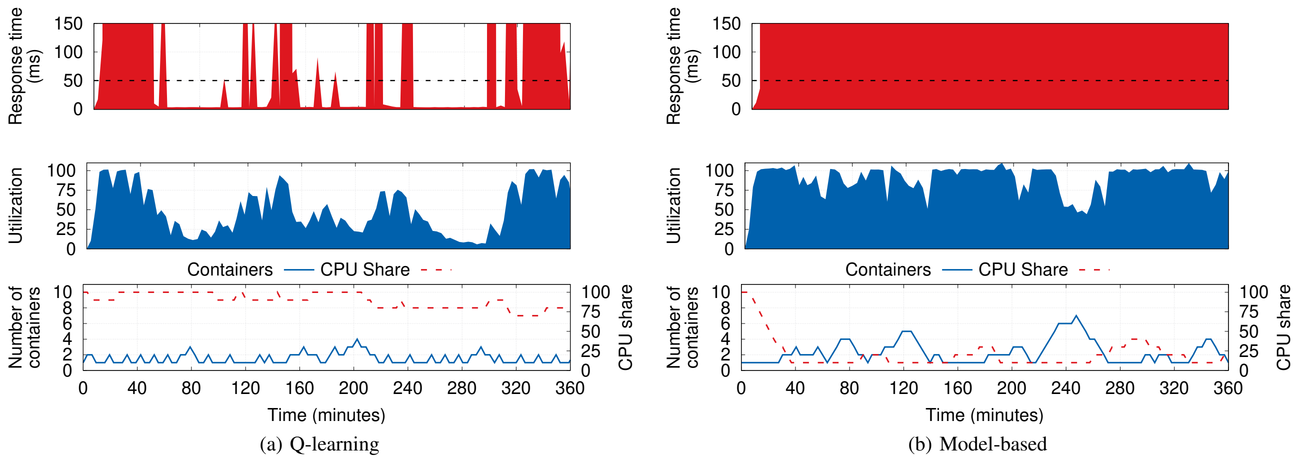
We first consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$: in this case, optimizing the application response time is more important than saving resources; moreover, adaptation costs are negligible. Q-learning slowly learns how to adapt the application deployment. As we can see from Fig. 4a and Table III, when the 5-action model is considered, Q-learning often changes the application deployment (i.e., 66% of the time) performing both horizontal and vertical scaling operations. Moreover, the application response time exceeds $R_{\max}$ for 30% of the time. Taking advantage of the system knowledge, the model-based solution reduces the number of $R_{\max}$ violations to 12% (see Fig. 4b). Differently from Q-learning, the model-based policy uses a higher number of medium-size containers: on average, it deploys 5.11 containers that can access to 47% of CPU (instead of 1.7 containers with 89% of CPU). In general, the model-based solution obtains better performance than Q-learning and more quickly reacts to workload variations (see Fig. 4). From Table IV, we can observe that the 9-action model makes the learning process more challenging for Q-learning, which violates $R_{\max}$ for 81%



(a) Q-learning

(b) Model-based

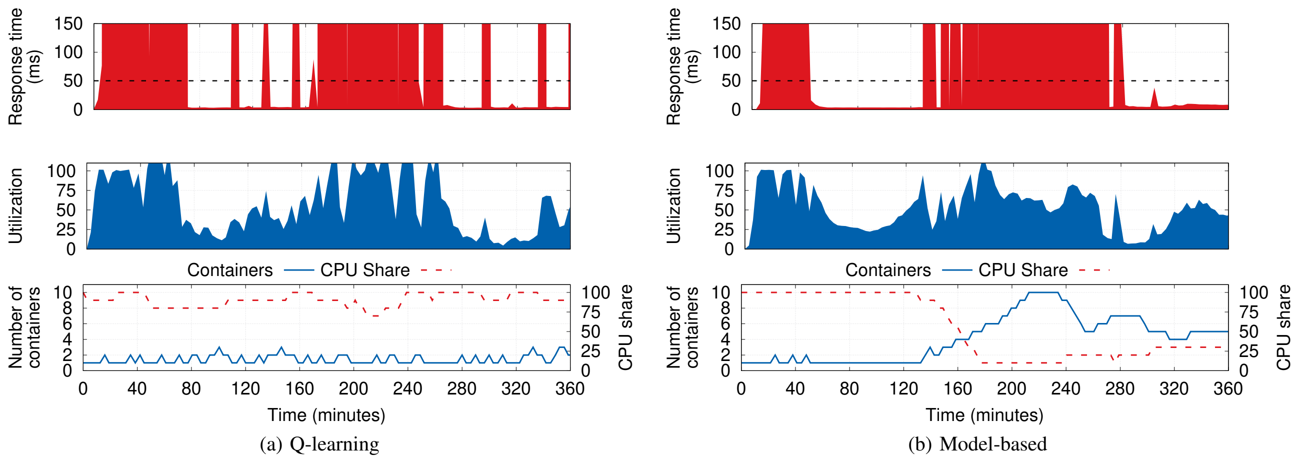
Fig. 4: Application performance using the 5-action adaptation model and weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$.



(a) Q-learning

(b) Model-based

Fig. 5: Application performance using the 5-action adaptation model and weights $w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$.



(a) Q-learning

(b) Model-based

Fig. 6: Application performance using the 5-action adaptation model and weights $w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$.

TABLE III: Prototype-based experiments: Application performance under different configurations of cost function weights and RL policies, when the 5-action adaptation model is used.

Weights	*Policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median* R (ms)	*Adaptations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**Q-learning**	29.77	48.81	88.70	1.70	4.18	65.65
	Model-based	**12.10**	37.35	47.34	5.11	6.11	50.81
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**Q-learning**	38.21	50.87	89.19	1.54	4.24	67.48
	Model-based	97.81	**90.85**	**24.01**	**2.12**	25959.45	61.31
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**Q-learning**	55.06	63.55	84.43	1.48	223.35	66.46
	Model-based	**39.23**	**52.22**	52.0	4.21	8.30	**32.31**

TABLE IV: Prototype-based experiments: Application performance under different configurations of cost function weights and RL policies, when the 9-action adaptation model is used.

Weights	*Policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median* R (ms)	*Adaptations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**Q-learning**	80.54	82.21	45.14	1.58	9738.49	93.51
	Model-based	**24.11**	31.53	46.03	7.10	6.92	44.68
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**Q-learning**	88.19	87.29	39.37	1.46	16587.03	90.55
	Model-based	97.58	**91.05**	**20.16**	**2.32**	29654.97	59.68
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**Q-learning**	96.77	94.07	40.40	1.49	18740.43	88.71
	Model-based	**33.60**	**60.19**	36.72	4.59	18.63	**28.0**

TABLE V: Prototype-based experiments: Application performance resulting from different configurations of static deployment.

Policy	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median* R (ms)	*Adaptations* (%)
Static with 10 containers and 100% CPU share	0.0	5.19	100.0	10.0	3.76	0.0
Static with 3 containers and 50% CPU share	0.0	29.45	50.0	3.0	4.29	0.0
Static with 2 containers and 50% CPU share	22.31	54.30	50.0	2.0	14.02	0.0

of the time (and registers 9.7 s as median response time). Also the model-based solution increases the number of $R_{\max}$ violations by 12% with respect to the 5-action model setting. However, the 9-action model allows to reduce the number of the deployment reconfigurations almost by 5% for model-based, also because this model allows both horizontal and vertical scaling in parallel.

We now consider the case when saving resources is more important than meeting the $R_{\max}$ bound and the adaptation costs are negligible, i.e., $w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$. Table III and IV show that, in general, Q-learning performs worse than model-based in terms of resource usage. Using the 5-action model, Q-learning deploys, on average, 1.54 containers that can use up to 89% of the CPU, leading to a low 51% of resource utilization. This is also visible in Fig. 5a. Conversely, the model-based solution identifies an adaptation policy that runs the application using, on average, 2.12 instances that access to 24% of the computing resources. As shown in Fig. 5b, the model-based algorithm produces a higher resource utilization (on average equal to 91%). Model-based has a similar behavior even when the 9-action model is used. On average, its resource usage is higher than in Q-learning (91.05% and 87.29%, respectively), as also the percentage of application response time $R_{\max}$ violations. This strictly follows from the weights configuration that clearly prefers to save resources in the face of $R_{\max}$ violations. As we can observe from Table IV, Q-learning finds a solution that is similar to that obtained by model-based. However, Q-learning changes continuously the application deployment (i.e., 90.55% of the time, instead of 59.68%).

When the different deployment goals are equally important, i.e., $w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$, the model-based solution reduces the number of $R_{\max}$ violations (39% instead of 55% by Q-learning, under the 5-action model, and 34% instead of 97%, under the 9-action model). Moreover, model-based changes the application deployment less than Q-learning; as a consequence, the application availability increases. From Fig. 6b, we can observe that model-based prefers horizontal scaling to vertical scaling. This depends on the definition of the adaptation cost in EDS: changing the container configuration (i.e., vertical scaling) is more expensive than adding new containers (i.e., horizontal scaling). Also under this configuration of weights, when the 9-action model is adopted, Q-learning registers a significant performance reduction. During the experiment, Q-learning cannot learn a suitable adaptation policy, resulting in a great number of reconfigurations (almost 89% of the time) and exceeding $R_{\max}$ for 96.77% of the time. Conversely, model-based obtains better performance in both the action configurations. In particular, the model-based solution exceeds $R_{\max}$ at most 34% of the time, under the 9-action model (39% with the 5-action model). Although it is not straightforward to draw conclusion on resource utilization, we can easily see from Tables III and IV that model-based identifies a trade-off configuration between those resulting from the previously considered weights configurations.

Discussion. Overall, the prototype-based experiments con-

firm the benefits of the model-based approach. From Table III and IV, we can appreciate the benefits of a model-based RL algorithm also with respect to a static configuration (reported in Table V). Although a static configuration can lead to satisfying performance for specific edge-case deployment goals, this configuration is application-specific and not flexible, meaning that it cannot react to sudden workload peaks or changes. Conversely, the RL-based approach is general and dynamic, requiring only to specify the deployment objectives to be pursued. It allows to specify *what* the user aims to obtain (through the cost function weights), instead of *how* it should be obtained. Among the RL approaches, the model-based is the most promising one, as also confirmed by the prototype-based experiments: it finds the best adaptation strategy for all weight and action model configurations.

We conclude by observing that the model-based solution is more computational demanding. Indeed, each learning update step requires to iterate over all the states, all the actions, and all the next states (see Eq. 3). The resulting computational complexity is $O(|\mathcal{S}|^2|\mathcal{A}|)$. However, given the limited number of available actions and that many transition probabilities are equal to 0, the complexity reduces to $O(K_{\max}\lceil \frac{1}{\bar{c}} \rceil \lceil \frac{1}{\bar{u}} \rceil^2)$.

VII. Conclusions

Most existing elasticity policies resort on threshold-based heuristics that require to express how specific goals should be achieved. In this paper, aiming to design more general and flexible solution, we have proposed different RL policies for controlling the elasticity of container-based applications. Specifically, we have designed and evaluated model-free and model-based solutions, which exploit different degree of knowledge about the system dynamics. We conducted a thorough evaluation of the different RL approaches using simulations and prototype-based experiments. To this end, we have developed EDS, an extension of Docker Swarm equipped with self-adaptation capabilities. The results have shown the flexibility and benefits of RL solutions: while Q-learning suffers from slow convergence time, the model-based approach can successfully learn the best adaptation policy, according to the user-defined deployment goals.

As future work, we will extend the proposed RL policies to control multi-component applications (e.g., microservices) deployed in a geo-distributed fog computing environment, where heterogeneous computing resources communicate with non-negligible network latencies. As regards the EDS architecture, we plan to design and integrate new decentralized control patterns for controlling the elasticity of container-based applications, e.g., recurring to a hierarchical control solution [10]. Indeed, the master-workers pattern still includes centralized components that can easily become a system bottleneck, especially when a multitude of applications, scattered in a large-scale geo-distributed environment, should be controlled. Conversely, a hierarchical (or a fully decentralized) approach can more efficiently control the application deployment, following the *divide-et-impera* principle. As such, it represents a promising approach for controlling microservice-based applications in fog environments.

References

[1] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, 2015.

[2] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *Proc. IEEE FiCloud '18*, 2018, pp. 85–92.

[3] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of Docker containers with ElasticDocker," in *Proc. IEEE CLOUD '17*, 2017, pp. 472–479.

[4] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," in *Proc. IEEE CLOUD '18*, 2018, pp. 82–89.

[5] M. Nardelli, V. Cardellini, and E. Casalicchio, "Multi-level elastic deployment of containerized applications in geo-distributed environments," in *Proc. IEEE FiCloud '18*, 2018, pp. 1–8.

[6] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable DevOps," in *Proc. SEAMS '17*, 2017, pp. 65–75.

[7] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proc. IEEE/ACM CCGrid '17*, 2017, pp. 64–73.

[8] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *Proc. IEEE ICAC '06*, 2006, pp. 65–73.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018.

[10] D. Weyns, B. Schmerl, V. Grassi, S. Malek *et al.*, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475, pp. 76–107.

[11] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Trans. Serv. Comput.*, vol. 11, pp. 430–447, 2018.

[12] X. Guan, X. Wan, B. Y. Choi, S. Song, and J. Zhu, "Application oriented dynamic resource allocation for data centers using Docker containers," *IEEE Commun. Lett.*, vol. 21, no. 3, pp. 504–507, 2017.

[13] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *Proc. ACM/SPEC ICPE '17 Comp.*, 2017, pp. 5–10.

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

[15] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, "Docker containers across multiple clouds and data centers," in *Proc. IEEE/ACM UCC '15*, 2015, pp. 368–371.

[16] Y. Mao, J. Oak, A. Pompili, D. Beer *et al.*, "DRAPS: dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster," in *Proc. IEEE IPCCC '17*, 2017.

[17] A. Asnaghi, M. Ferroni, and M. D. Santambrogio, "DockerCap: A software-level power capping orchestrator for Docker containers," in *Proc. IEEE EUC '16*, 2016, pp. 90–97.

[18] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Trans. Cloud Comput.*, in press.

[19] D. Elliott, C. Otero, M. Ridley, and X. Merino, "A cloud-agnostic container orchestrator for improving interoperability," in *Proc. IEEE CLOUD '18*, 2018, pp. 958–961.

[20] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, 2018.

[21] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Trans. Serv. Comput.*, in press.

[22] "Swarm mode overview," https://docs.docker.com/engine/swarm/, 2019.

[23] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[24] Z. Jerzak and H. Ziekow, "The DEBS 2015 grand challenge," in *Proc. ACM DEBS 2015*, 2015, pp. 266–268.