Dynamic Multi-metric Thresholds for Scaling Applications Using Reinforcement Learning

Fabiana Rossi, *Member, IEEE*, Valeria Cardellini, *Member, IEEE,*
Francesco Lo Presti, *Senior Member, IEEE,* and Matteo Nardelli

Abstract—Cloud-native applications increasingly adopt the microservices architecture, which favors elasticity to satisfy the application performance requirements in face of variable workloads. To simplify the elasticity management, the trend is to create an auto-scaler instance per microservice, which controls its horizontal scalability by using the classic threshold-based policy. Although easy to implement, setting manually the scaling thresholds, which are usually statically-defined on a single metric, may lead to poor scaling decisions when applications are heterogeneous in terms of resource consumption.
In this paper, we study dynamic multi-metric threshold-based scaling policies, that exploit Reinforcement Learning (RL) to autonomously update the scaling thresholds, one per controlled resource (CPU and memory). The proposed RL approaches (i.e., QL, MB, and DQL Threshold) use different degrees of knowledge about the system dynamics. To model the thresholds' adaptation actions, we consider two RL-based architectures. In the single-agent architecture, one agent drives the updates of both scaling thresholds. To speed-up the learning, the multi-agent architecture adopts a distinct agent per threshold. Simulation- and prototype-based results show the benefits of the proposed solutions when compared to the state-of-the-art policies and highlight the advantages of multi-agent MB Threshold and DQL Threshold approaches, in terms of deployment objectives and execution times.

Index Terms—Elasticity, Self-adaptation, Reinforcement Learning, Deep Q-Learning, Microservice Architecture.

1 Introduction

Cloud computing has encouraged the development of elastic applications, whose deployment can be adapted at runtime to meet application-level Quality of Service (QoS) requirements in face of changing operating conditions. Moreover, many large enterprises (e.g., Amazon, Netflix, Spotify) have reshaped their applications from monolithic to microservice architectures, so to improve efficiency and scalability of applications. The microservice architecture splits an application into many autonomous, fine-grained, and loosely coupled services, each providing a specific and independent functionality. A microservice is typically deployed using software containers, which bound together the application code and its dependencies (e.g., libraries), improving portability. To simplify the deployment and runtime management of containers, orchestration engines are adopted (e.g., Kubernetes, Docker Swarm, Amazon ECS). They allow to create multiple, decentralized auto-scaler instances, one for each microservice to deploy.

A variety of methodologies can be applied to control the microservice elasticity, as surveyed in [1], [2]; among them, threshold-based policies are the most popular solution thanks to their simplicity. According to this class of policies, the number of microservice replicas is increased (or decreased) as soon as a relevant metric is above (or below) the scale-out (or scale-in) threshold value. Although this approach is easy to implement and scales well, it moves complexity from designing the reconfiguration strategy to selecting critical values for the thresholds. In addition, most threshold-based auto-scalers consider a statically-defined threshold on a single resource metric (e.g., CPU utilization). Since today's applications are heterogeneous in nature (e.g., [3], [4]), being CPU, memory intensive, or mixed, the existing policies may result in a poor scaling strategy. A further challenge arises from the user requirement of specifying a Service Level Objective (SLO) based on user-oriented metrics (e.g., response time, throughput, monetary cost) rather than configuring system-oriented metrics as common scaling thresholds are.

Differently from the popular static threshold-based approaches, our aim is to design a flexible solution that can dynamically adapt the scaling thresholds without the need of manual tuning. We want to make use of experience to learn how to efficiently update the application deployment, also without defining an exact model of the microservice behavior. To this end, we resort to Reinforcement Learning (RL), 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. RL allows to express *what* the user aims to obtain, instead of *how* it should be obtained (as done by static threshold-based policies). In [5], we started to explore RL approaches for driving application elasticity. The proposed model explicitly controlled the number of replicas. Although this choice appears to be intuitive, it does not lend itself to simple implementations on current container orchestration frameworks, which have been designed to control elasticity through threshold-based mechanisms.

• *F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli are with University of Rome Tor Vergata, Italy. E-mails: {f.rossi, cardellini, nardelli}@ing.uniroma2.it, lopresti@info.uniroma2.it*

Moreover, with this approach, the RL agent learns a policy that autonomously performs the scaling, which might not always result from load peaks, response time violations, or other changes in the monitored performance metrics.

In this paper, we present dynamic (or self-adaptive) threshold-based scaling policies that can automatically learn and update the values of multiple scaling thresholds, one for each metric of interest. Building on our previous work [6], where we considered only CPU utilization, we now generalize our result by proposing different approaches to control multiple metrics (i.e., CPU and memory utilization) and to reduce the policy execution time. The proposed approach moves towards the direction of an explainable scaling policy, where the RL agent learns a policy that can be intuitively understood (and constrained, as well) by means of the thresholds. Such an approach can also pave the way for the adoption of RL solutions in current systems: by letting each agent learn the correct threshold by itself, we can get past today's threshold handcrafting. First, we present a single-agent solution, where a single RL agent is in charge of updating multiple scaling thresholds. To improve RL scalability, we then design a multi-agent solution where each RL agent oversees a specific resource metric. In this general framework, we design and evaluate model-free and model-based RL algorithms; they exploit different degrees of knowledge about the system dynamics to find a trade-off between system and user-oriented metrics (i.e., resource utilization and target response time). As model-free policies, we propose QL Threshold and DQL Threshold, two dynamic threshold-based policies built on Q-learning and Deep Q-learning, respectively. Deep Q-learning approximates the system knowledge by using a deep neural network. Since model-free solutions can suffer from slow convergence rate, we then provide the learner with basic knowledge about its environment, so to boost the learning process. Hence, we propose MB Threshold, an approach that exploits what is known about the system dynamics to update the scaling thresholds. We have extensively evaluated the benefits of dynamic multi-threshold policies, via simulation and prototype evaluation. To this end, we have implemented all the proposed solutions into Kubernetes. Through simulations, we evaluate the proposed approaches under different deployment objectives and workloads, using real-world traces (i.e., NYC Taxi Ride and Bitbrains). Using Kubernetes, we evaluate the designed solutions in a real environment, and show the advantages of the dynamic threshold-based approach over Kubernetes' default scaling policy and two state-of-the-art solutions (i.e., [7], [8]).

The remainder of the paper is organized as follows. We discuss related works in Section 2. We then present the system model and the idea of a dynamic multi-metric threshold policy based on RL (Section 3). In Section 4, we propose the single-agent and multi-agent RL architectures, where the latter aims to simplify the policy design and reduce its execution time. Section 5 details the model-free and model-based approaches to update the scaling thresholds. Then, we evaluate the proposed solutions in Sections 5.3 and 6 and conclude in Section 7.

2 Related Work

We focus on the methodologies proposed in literature for the elasticity of applications in cloud environments, considering recent works with respect to existing surveys (e.g., [1], [2]). The methodology identifies the class of algorithms used to plan the deployment adaptation so to achieve specific goals. Existing elasticity solutions rely on a broader set of methodologies, that we classify in the following categories: mathematical programming, control theory, queuing theory, machine learning, and threshold-based solutions.

Mathematical programming approaches exploit tools from operational research in order to change the microservice parallelism degree. The formulation and resolution of Integer Programming (IP) problems belongs to this category. IP formulations have been mainly used to solve the placement of application instances (e.g., [9]). However, some works consider these approaches also to address the application elasticity problem (e.g., [10], [11], [12]). For example, Rahman et al. [11] solve an optimization problem to find the target CPU utilization values (i.e., thresholds) to automatically scale microservices based on the predicted application response time. The main drawback of mathematical programming approaches is scalability: since the deployment problem is NP-hard, resolving the exact formulation may require prohibitive time as the problem size grows.

Few research works rely on control theory to scale applications. In this case, the policy usually identifies three main entities: disturbance, decision variables, and system configuration. Disturbances represent events that cannot be controlled; nevertheless, their future value can be predicted (at least in the short term), e.g., incoming data rate, load distribution, and processing time. The decision variables identify the replication of each application component. Shevtsov et al. [13] show that, although research on control-theoretical software adaptation is still in its early stages, an ever increasing number of solutions has recently consider control theory to realize self-adaptive systems (e.g., [14], [15], [16]). Baresi et al. [14] combine infrastructure and application-level adaptation using a discrete-time feedback controller. It considers the application response time as a function of the assigned CPU cores (decision variables) and the request rate (disturbance) for horizontally and vertically scaling applications. The critical point of control-theoretic approaches is the need of a good system model, which can sometimes be difficult to formulate, e.g., when the decision variables inter-play in a complex manner.

Queuing theory is often used to predict the response time of a microservice with respect to its replication degree. The key idea is to model the microservice as a queuing system with inter-arrival and service times having general statistical distributions. In general, queuing theory is well suited to determine the replication degree of microservices or predict performance (e.g., [17], [18], [19]). Nevertheless, it often requires to approximate the system behavior so to apply models from the established theory. Therefore, when the system is very complex, also the queuing theory becomes complex, discouraging its adoption.

In the last years, different research works have exploited the advances of machine learning approaches to model functions based on observed data (e.g., resource utilization,

application performance) or to dynamically adapt the application deployment. Islam et al. [20] utilize a combination of deep neural networks and linear regression to predict the future resource demands. To proactively scale containers in response to workload changes, Imdoukh et al. [21] use deep neural networks to learn a container elasticity policy that considers past scaling decisions and workload behavior. Yu et al. [22] propose Microscaler, a framework that relies on Bayesian Optimization and a step-by-step heuristic for the adaptive deployment of microservices-based applications. Although conceptually easy to design, machine learning techniques may suffer from two main drawbacks. First, they require reasonably large training sets. Second, they cannot easily and rapidly address unforeseen configurations.

Reinforcement Learning is a special method belonging to the branch of machine learning [23]. RL techniques can learn an adaptation policy through a trial-and-error process. After executing an action in a specific system state, the RL agent observes the cost experienced by the system, so to learn how good the performed action was. The obtained cost contributes to update the lookup table that stores the estimation of the long-term cost for each state-action pair. The RL agent must try a variety of actions and progressively favor those that appear to be the best ones. Several works have exploited RL techniques to drive elasticity in cloud computing, as surveyed in [1]. However, most of them consider model-free RL algorithms, such as Q-learning and SARSA (e.g., [8], [24], [25], [26]), which suffer from slow learning rate, as observed in [27]. To overcome this issue, different model-based RL approaches have been proposed. They use a model of the system to drive the action exploration and speed-up the learning phase. Tesauro et al. [28] combine RL with queuing network, whereas Arabnejad et al. [24] consider a fuzzy inference system to model the application performance. In [5], we propose a model-based approach that boils down to replacing the model-free equation to update the lookup table with one step of the value iteration algorithm using empirical estimates for the unknown parameters. Although model-based RL approaches can overcome the slow convergence rate of model-free solutions, they can suffer from poor scalability in systems with large state space, because the lookup table has to store a separate value for each state-action pair. An approach to overcome this issue consists in approximating the system state or the action-value function, so that the agent can explore a reduced number of system configurations. Deep Q-learning has recently been used to approximate the system knowledge; it integrates deep neural networks into RL [29]. Thereafter, it has been widely applied in a variety of domains, e.g., container migration [30], traffic offloading [31], and device placement [32]. However, to the best of our knowledge, Deep Q-learning has been little applied in the context of elastic resource provisioning (e.g., [33], [34]).

Many solutions exploit best-effort threshold-based policies, based on the definition of static thresholds for scaling in/out microservices at run-time. Most of the research works (e.g., [35], [36]) and some Cloud service providers (e.g., Amazon, IBM) use a single threshold value for driving elasticity. Although it often works sufficiently well, most existing applications are heterogeneous in nature, implying that it is not enough considering only one aspect of resource metrics to scale. Google's Compute Engine auto-scaling supports multiple metrics and corresponding scaling rules and picks the metric that results in the largest number of virtual machines (VMs) [37]. In the context of microservices, few works [7], [38] take into account CPU, memory, and network utilization to adapt at run-time the application deployment. In particular, HyScale [7] is a threshold-based auto-scaler that considers CPU and memory utilization. Although threshold-based scaling is easy to implement, it is a best-effort approach that provides no guarantees about the reconfiguration optimality. Furthermore, selecting the threshold values can be challenging, especially in case of multiple metrics.

To overcome such a manual setting of thresholds, self-adaptive (or dynamic) threshold-based policies have been proposed in different contexts, e.g., performance management, VM consolidation (e.g., [8], [39], [40]). To the best of our knowledge, they have been scarcely applied for scaling applications (e.g., [6], [8]). Horovitz et al. [8] rely on Q-learning and a heuristic to automatically learn and adapt the scaling thresholds. Nevertheless, they consider only homogeneous microservices and a single scaling metric. In [6], we propose a two-layered hierarchical solution to control the elasticity of microservice applications. Having a global knowledge of the application, a high-level centralized entity coordinates the microservices' scaling decisions (global policy). At low level, decentralized entities locally control the adaptation of single microservices using RL (local policy). However, only CPU-bound applications are considered and an analysis on the local policy scalability is missing. We believe that microservices heterogeneity and time required to promptly react to sudden workload changes are critical factors in a dynamic environment. This paper extends [6] by focusing on local scaling policies. Differently from all the above works, we propose a dynamic and multi-metric threshold-based policy that relies on RL to set the thresholds values. To improve the RL scalability when the number of scaling thresholds increases, we investigate two different solutions: first, we integrate deep neural networks into RL; then, we evaluate a multi-agent approach. For our prototype-based experiments, we use HyScale [7], the policy by Horovitz et al. [8], and the Kubernetes' threshold-based policy as baselines to compare against our approach.

3 Self-adaptive Threshold-based Policy

In this section, first we present the elasticity problem when heterogeneous microservices are considered. Then, we formally introduce the basic concepts of RL and illustrate how to use RL to adapt the scaling thresholds at run-time.

3.1 Problem Definition and System Model

We consider a general microservice model, where a microservice is a black-box entity that carries out a specific task (e.g., performs computation, accesses data sets). To efficiently handle varying workloads and meet QoS requirements, the amount of computing resources granted to the microservice should be dynamically adjusted. To this end, we can exploit horizontal elasticity which allows to increase (scale-out) and decrease (scale-in) the number of

microservice replicas according to a deployment policy. The latter should properly drive the microservice elasticity so to guarantee the desirable performance, while minimizing the monetary cost. In this work, we consider latency-sensitive microservices that expose a SLO defined as target response time that should not be exceeded (i.e., $T_{\max}$). By exploiting horizontal elasticity, microservice replicas can process incoming requests in parallel, thus reducing the per-replica load and, in turn, the processing latency.

To manage the microservices auto-scaling over time, we need a deployment controller (or auto-scaler) that provides self-adaptation mechanisms and can be equipped with deployment policies. Today's cloud providers that support multi-component applications allow to create multiple, decentralized auto-scaler instances, each carrying out the adaptation of a single microservice deployment. By periodically analyzing the collected data about the microservice and the execution environment, the auto-scaler determines whether the microservice deployment should be changed. To determine the microservice scaling actions, most of the existing auto-scalers rely on a static threshold-based policy, where the scaling threshold is usually set on a specific resource metric (i.e., CPU or memory). Despite its simplicity, the use of single metric might not be well suited for all the scenarios, e.g., applying CPU utilization to scale a memory-bound microservice, since in this case the microservice performance is expected to be determined by the memory usage rather than the CPU utilization. To address this issue, we define one scaling threshold for each relevant metric and, for the sake of simplicity, consider CPU utilization and memory usage. We define as u the microservice CPU utilization and as r the amount of used memory. These metrics are defined as the ratio between the monitored resource demand and the amount of assigned resource to the microservice. A microservice cannot use more resources than the configured limit. We introduce the CPU and memory scale-out thresholds, denoted as $_u$ and $_r$ respectively, and the CPU and memory scale-in thresholds, as $^{\text{in}}_u$ and $^{\text{in}}_r$. To simplify the setting of multiple scaling thresholds, we design a dynamic threshold-based scaling policy that automatically learns and updates their values using RL.

3.2 Reinforcement Learning-based Policy

RL is a machine learning technique where an agent learns the optimal policy through direct interaction with the system. There are three basic concepts in RL: *state*, *action*, and *cost*. The state describes a system configuration. The action is what a RL agent can do in a given state. The cost is an immediate feedback that a RL agent receives when it performs an action in a state. Intuitively, the cost allows the RL agent to discriminate between good and bad system configurations and actions. A RL agent aims to learn an optimal adaptation strategy (i.e., state-action mapping), so to minimize a discounted cost over an infinite horizon.

To update the scaling thresholds, we consider that the RL agents interact with the microservices in discrete time steps. Learning by experience, a RL agent estimates the relationship between application and system-oriented metrics, and accordingly adapts the scale-out thresholds (i.e., $_u$ and $_r$). In this work, we do not dynamically update the scale-in thresholds (i.e., $^{\text{in}}_u$ and $^{\text{in}}_r$); nevertheless, our approach can be easily extended to account for them. We denote by S the set of all the microservice states. We model a microservice state as a tuple of N relevant features. Unlike most of the literature where the state considers the number of replicas (e.g., [5], [24], [28]), we define it in terms of thresholds; this choice is specifically tailored for existing orchestrators and thus is better suited for ease of integration with them. For each state $s \in S$, we have a set of *feasible* adaptation actions $A(s) \subseteq A$, where A is the set of all actions. At each time step, the agent observes the microservice state $s \in S$ and, according to an action selection policy, performs an action $a \in A(s)$ to update the scaling threshold. One time step later, the microservice transits in a new state $s' \in S$, experiencing the payment of an immediate cost $c(s;a;s') \in \mathbb{R}$. Both the paid cost and the next state transition are stochastic, because they usually depend on external unknown factors. In our model, the RL agent wants to minimize the cost so to jointly satisfy microservice performance and limit resource wastage. We model the cost function so to include two different contributions: the performance penalty c_{perf} and the resource monetary cost c_{res}. The performance penalty takes into account performance degradation while the monetary cost accounts for resource wastage. Formally, we define the immediate cost function $c(s;a;s')$ as a weighted sum of the normalized costs:

$$c(s;a;s') = w_{\text{perf}}\ c_{\text{perf}}(s;a;s') + w_{\text{res}}\ c_{\text{res}}(s;a;s') \quad (1)$$

where w_{perf} and w_{res}, $w_{\text{perf}} + w_{\text{res}} = 1$, are non-negative weights that allow us to express the relative importance of each cost term. By observing the incurred immediate cost, the agent updates and estimates the so-called Q-function, thus improving the threshold update policy. The Q-function consists in $Q(s;a)$ terms, which represent the expected long-term cost that follows the execution of action a in state s. So, it drives the scaling threshold updates.

Figure 1 presents a high-level overview of the two different approaches we propose to support multiple scaling thresholds, with one threshold for each relevant performance metric (i.e., CPU and memory). First, we consider a *single-agent* architecture, where a single RL agent updates the scaling thresholds by taking into account the estimated contribution of multiple metrics to the application performance (Section 4.1). Then, we present a *multi-agent* architecture, which defines multiple RL agents where each oversees the threshold on a specific resource metric (Section 4.2).

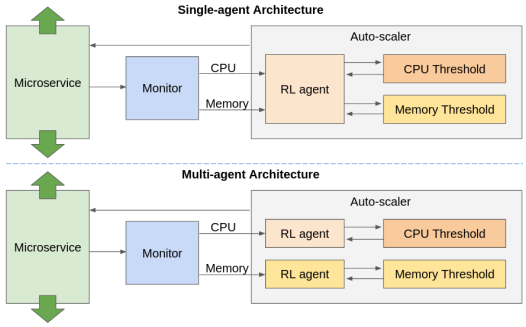


Fig. 1. High-level overview of the single-agent and multi-agent solution.

4 RL-BASED ARCHITECTURES

In this section, we present the single and multi-agent architectures, which use different state and action models for driving the scale-out thresholds updates. For both approaches, the Q-function is updated using three different RL methods (Q-learning, Model-based, and Deep Q-learning), as described in Section 5.

4.1 Single-agent RL

The single-agent (SA) architecture uses a single RL agent per microservice to control the scale-out thresholds for the two performance metrics, i.e., CPU and memory utilization. We believe this is the most intuitive approach, especially if we consider the RL agent as a black-box entity that will eventually learn a suitable threshold update strategy.

State. For the monitored microservice, at time i, we define its state as $s_i = (\ _u; u;\ _r; r)$, where u is the CPU utilization, r is the memory utilization, and $_u$ and $_r$ are the corresponding scale-out thresholds. Being the CPU utilization, u, a real number in $[0; 1]$, we discretize it by assuming that $u \in \{0; u; \ldots; L_u u\}$, where u is a suitable quantum and $L_u \in \mathbb{N}$ such that $L_u u = 1$. Similarly, we discretize the memory utilization $r \in \{0; r; \ldots; L_r r\}$, where r is a suitable quantum and $L_r \in \mathbb{N}$ such that $L_r r = 1$. We also assume that $_u$ and $_r$ range in the interval $[\ _{\min};\ _{\max}]$, with $0 < \ _{\min} \ \ _{\max} < 1$.

Action. In a state s, the RL agent identifies the threshold adaptation action to be performed. We propose an action model that consists of $A = \{\ _u;\ _r; 0;\ _u;\ _r\}$, where $_u$, $_r \in (0; 1)$ are suitable threshold quanta. In particular, $a = 0$ is the *do nothing* decision, whereas $\ _u$ (or $\ _r$) represents a threshold adaptation action: $+\ _u$ (or $+\ _r$) to add a CPU (or memory) threshold quantum and $\ _u$ (or $\ _r$) to decrease by a threshold quantum. We observe that the number of microservice states $|S|$ depends also on the threshold quanta values: $|S| = (L_u + 1)\ (L_r + 1)\ (\frac{\ _{\max}}{\ _u} + 1)\ (\frac{\ _{\max}}{\ _r} + 1)$.

Cost. To each triple $(s; a; s')$ we associate an immediate cost function $c(s; a; s')$, which combines the performance c_{perf} and resource cost c_{res} as in (1). This cost is aimed to provide a feedback to the RL agent during the learning stages. In a single-agent setting, the scaling threshold update depends on multiple system-oriented metrics. For this reason, we distinguish two contributions for the performance penalty: the former is related to CPU resource, the latter to memory resource. The CPU performance penalty $c_{\text{perf};u}$ is paid whenever the microservice response time t approaches (or exceeds) the SLO response time $T_{\max}$, because higher CPU utilization causes higher response times (e.g., see [41], [42]). Similarly, the memory performance penalty $c_{\text{perf};r}$ is paid when the used memory d exceeds the assigned memory m. In this case, the microservice incurs in an out-of-memory exception, which causes performance penalties and possibly data loss in real applications. We have:

$$c_{\text{perf};u}(s; a; s') = \begin{cases} e^{\ \frac{t' - T_{\max}}{T_{\max}}} & t' \geq T_{\max} \\ 1 & \text{otherwise} \end{cases}$$
$$c_{\text{perf};r}(s; a; s') = \begin{cases} e^{\ \frac{d' - m'}{m'}} & d' \geq m' \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

being $$ a constant that determines the exponential function steepness, t', d', and m' the microservice response time, the used memory, and the assigned memory, respectively, in s'. The received cost grows faster as the monitored metric approaches its target value. Since the critical resource drives the thresholds update, we compute $c_{\text{perf}} = \max\{c_{\text{perf};u}; c_{\text{perf};r}\}$. Also for the resource cost, we distinguish two different contributions, $c_{\text{res};u}$ for CPU utilization and $c_{\text{res};r}$ for memory usage. For each metric, we can reasonable assume that the resource cost increases when the scale-out threshold decreases, because the lower the scale-out threshold, the higher the number of used resources. Formally, we define $c_{\text{res};u}$ and $c_{\text{res};r}$ as:

$$c_{\text{res};u}(s; a; s') = e^{\ \frac{\ _{\min} - \ '_u}{\ _{\min}}}$$
$$c_{\text{res};r}(s; a; s') = e^{\ \frac{\ _{\min} - \ '_r}{\ _{\min}}} \quad (3)$$

being $$ a constant that determines the exponential function steepness, $\ '_u$ and $\ '_r$ the scale-out thresholds in s'. The exponential function allows the performance and resource cost to have the same dynamics, making them more easily comparable with one another. Note that the resource cost does not explicitly consider resource usage, which is not accounted for in our microservice state s; this relates to the cost definition as known and unknown cost (as will be presented in Section 5.2). Defining c_{res} as a function of the threshold, which is explicitly accounted for in the system state s, allows us to consider it as a known cost, thus reducing the amount of unknown cost to be estimated through experience. This approximation improves the learning process for model-based solutions. At each discrete time step, we set $c_{\text{res}} = \max\{c_{\text{res};u}; c_{\text{res};r}\}$. By observing the incurred immediate costs, the Q-function is updated over time, thus improving the threshold update policy.

4.2 Multi-agent RL

The multi-agent (MA) approach is a solution for improving the scalability of an agent that needs to control multiple metrics. It uses simplifying assumptions leading to independent RL agents for each monitored metric that exploit a more compact state space. We use a RL agent for updating the CPU threshold, referred as u-agent, and a RL agent to update the memory threshold, referred as r-agent. These agents operate in parallel to update the scaling thresholds.

State. For the u-agent, we define $s = (\ _u; u)$, where $_u$ is the CPU utilization scale-out threshold, and u is the CPU utilization. In this case, the state space cardinality is $|S| = (L_u + 1)\ (\frac{\ _{\max}}{\ _u} + 1)$. Similarly, for the r-agent, $s = (\ _r; r)$, where $_r$ is the memory utilization scale-out threshold and r is the amount of used memory.

Action. For both the agents, we propose an action model that consists of $A = \{\ ; 0;\ \}$, where $\in (0; 1)$ is a threshold quantum to add or subtract to the managed threshold.

Cost. The execution of a in s leads to the transition in a new microservice state s' and to the payment of an immediate cost. We define the immediate cost $c(s; a; s')$ as a weighted sum of c_{perf} and c_{res}, according to (1). The u-agent defines c_{perf} as $c_{\text{perf,u}}$ of (2) and c_{res} as $c_{\text{res,u}}$ of (3). Similarly, the r-agent defines c_{perf} as $c_{\text{perf, r}}$ and c_{res} as $c_{\text{res, r}}$. The observed immediate cost allows each agent to update its Q-function over time.

5 Q-function Update Strategies

Existing RL policies range from model-free to model-based solutions, according to the degree of system knowledge exploited to approximate the system behavior [23]. A model-free solution requires no a-priori knowledge of the system dynamics, which are learnt over time by interaction with the system. A model-free RL agent must prefer actions that it found to be effective in the past (*exploitation*). However, to discover such actions, it has to explore new actions (*exploration*). One of the main challenges of model-free RL agents is to find a good trade-off between exploration and exploitation. Differently, a model-based technique enriches the RL agent with a model of the system, which drives the exploration actions and speeds up the learning phase. As a downside, it may limit the agent scalability. We consider three different RL methods that differ on how they estimate and update the Q-function. First, we consider the simple model-free Q-learning algorithm. Then, we present a model-based approach, which exploits the known (or estimated) system dynamics to accordingly update the Q-function. Both these learning solutions explicitly use a $|S| \times |A|$ table (i.e., Q-table) to store the Q-value for each state-action pair. The Q-table allows to store the real experience without approximation. However, this approach may suffer from slow convergence rate when the number of state-action pairs increases. To tackle this issue, we also present a Deep Q-learning approach that combines Q-learning with deep neural networks. The neural network allows to approximate the Q-function using a non-linear function; in such a way, the RL agent can explore a reduced number of system configurations before learning a good adaptation policy.

5.1 Q-learning Threshold (QL Threshold)

In QL Threshold, the RL agent uses the model-free Q-learning to update the scaling threshold. At time i, the Q-learning agent selects action a_i to perform in state s_i using an ε-greedy policy on $Q(s_i; a_i)$; the microservice transits in s_{i+1} and experiences an immediate cost c_i. The ε-greedy policy selects the best known action for a particular state (i.e., $a_i = argmin_{a \in A(s_i)} Q(s_i; a)$) with probability $1 - \varepsilon$, whereas it favors the exploration of sub-optimal actions with low probability, ε. At the end of each time slot i, $Q(s_i; a_i)$ is updated in $O(1)$ using a weighted average:

$$Q(s_i; a_i) \leftarrow (1 - \alpha) Q(s_i; a_i) + \alpha \left(c_i + \min_{a' \in A(s_{i+1})} Q(s_{i+1}; a') \right)$$

where $\alpha \in [0; 1]$ is the *learning rate* parameter and $\gamma \in [0; 1)$ is the *discount factor*.

5.2 Model-Based Threshold (MB Threshold)

MB Threshold builds on a model-based RL agent to update the scaling threshold. A model-based RL agent exploits a system model to speed up the learning phase. Differently from model-free solutions, it does not use an action selection policy, but it always selects the best action in terms of Q-value, i.e., at time i, $a_i = argmin_{a \in A(s_i)} Q(s_i; a)$. Moreover, the model-based RL approach replaces the model-free equation to update the Q-function with the Bellman equation:

$$Q(s; a) = \sum_{s' \in S} p(s'|s; a) \left[c(s; a; s') + \min_{a' \in A(s')} Q(s'; a') \right] \quad \forall s \in S, \forall a \in A(s) \tag{4}$$

where we use estimations for the unknown parameters of the transition probabilities $p(s'|s; a)$ and the cost function $c(s; a; s')$, $\forall s; s' \in S$. Iterating over all states, actions, and next states, the update is performed in $O(|S|^2|A|)$. At time i, we estimate $p(s'|s; a)$ as the relative number of times the system transits from state s to s' given action a in the time interval $\{1; \ldots; i\}$. To better explain how to estimate $p(s'|s; a)$, we consider the u-agent of the multi-agent. We estimate $p(s'|s; a)$ as the relative number of times the CPU utilization changes from state u to u', given a, in the time interval $\{1; \ldots; i\}$. Similar arguments apply for the RL agents that resort on a different system state definition.

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 resource costs. The *unknown cost* $c_u(s')$ depends on the next state s'; it accounts for the performance penalty (1). As we assume that the application model is not known, we have to estimate $c_u(s')$ at run-time. Therefore, at time i, the RL agent observes the immediate cost c_i, computes $c_{u;i}(s') = c_i - c_{k;i}(s; a)$, and updates the estimate of the unknown cost $\hat{c}_{u;i}(s')$ as:

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

where $\beta \in [0; 1]$ is the *smoothing factor*. $\hat{c}_{u;i}(s')$ is then used to compute the cost of applying a in s according to (5).

For the estimates of the unknown cost, given a state s and the next state s', we observe that the expected performance penalty is not lower when the scale-out threshold or the resource utilization increases. Vice versa is also true. Considering the u-agent of the multi-agent solution, to speed-up the learning phase, we can heuristically enforce the following properties while updating $\hat{c}_{u;i}(s)$, $\forall s \in S$:

$$\hat{c}_{u;i}(s) \leftarrow \hat{c}_{u;i}(s') \quad \forall u' \le u; u' \ne u$$
$$\hat{c}_{u;i}(s) \leftarrow \hat{c}_{u;i}(s') \quad \forall u' \ge u; u' \ne u$$

Similar arguments apply to the other state definitions.

5.3 Deep Q-learning Threshold (DQL Threshold)

Deep Q-learning (DQL) Threshold is a Q-learning algorithm that uses a multi-layered neural network to approximate the Q-function. So, the network is also called Q-network. For a state space S and an action space A containing $|A|$ actions, the Q-network is a parametrized function from $\mathbb{R}^N$ to $\mathbb{R}^{|A|}$. In a given state s, the Q-network outputs a vector of action values $Q(s; \cdot; \theta)$, where θ are the network parameters. At each discrete time step i, the RL agent performs an action a in the state s, so the microservice transits in s' and observes the immediate cost c. At this point, the Q-network is updated by performing a gradient-descent step on $(y_i - Q(s; a; \theta_i))^2$ with respect to the network parameters θ_i, where y_i is the estimated long-term cost, defined as $y_i = c + \gamma \min_{a'} Q(s'; a'; \theta_i)$. When only the current

experience (i.e., $(s; a; c; s^0)$) is considered, this approach is too slow for practical real scenarios. Moreover, it is unstable due to correlations existing in the sequence of observations. To overcome these issues, Mnih et al. [29] revised the classic DQL algorithm introducing two major changes: the use of a replay buffer and a separate target network to compute y_i. At run-time, the DQL agent observes the application state and selects an adaptation action using the estimates of Q-values, as Q-learning does. To perform experience replay, at each time step i, the agent stores its experience $e_i = (s_i; a_i; c_i; s_{i+1})$ in a buffer D_i with finite capacity. At the end of the time step i, DQL resorts on samples (or mini-batches) of the experience $(s; a; c; s^0)$ $U(D_i)$ to simulate the interaction between the service and the environment, and accordingly update the Q-network. The mini-batch of experience $U(D_i)$ is drawn uniformly at random from the pool of samples D_i to remove correlations in the observation sequence and to smooth over changes in the data distribution. In standard DQL, the same Q-network is used both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic estimates. To prevent this, we decouple the action selection from its evaluation. In this approach, two networks are used (online and target network) and two value functions are learned. This results in two sets of weights, and , where the first is used to determine the greedy policy and the second to determine its value. At iteration i, the Q-network update uses the following loss function:

$$L(_i) = \sum_{U(D_i)} (c + \quad min_{a^0} Q(s^0; a^0; \ _i) \quad Q(s; a; \ _i))^2 \quad (7)$$

where 2 $(0; 1)$ is the discount factor, $_i$ are the online Q-network parameters at iteration i, and $_i$ are the target network parameters at iteration i. The $_i$ parameters are updated to the $_i$ values only every steps and are held fixed between individual updates. The loss function is optimized by stochastic gradient descent; its computational complexity depends on the neural network structure.

Note that DQL is model-free: it solves the RL task directly using samples, without explicitly estimating the cost function and the transition probabilities. It is also off-policy, because it learns a policy that is different from the policy used for action selection, which ensures adequate exploration of the state space. We resort to an -greedy policy as action selection policy.

We evaluate the proposed solutions using simulation experiments. We show the flexibility of the RL-based approaches under different cost function configurations (Section 5.5). Then, we compare the multi-agent approach against the single-agent one (Section 5.6). In Section 5.7, we generalize the evaluation by considering different workloads.

5.4 Experimental Setup

We consider a reference microservice application modeled as an M/M/k_i queue, where k_i is the number of microservice replicas at time step i. Each microservice replica has a service rate of 120 requests/s; the SLO requires the application response time to be at most $T_{\max} = 12$ ms. The scale-in threshold is set to 20% of CPU and memory

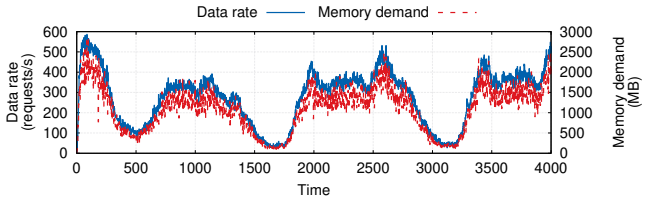


Fig. 2. Workload based on NYC Taxi Ride traces.

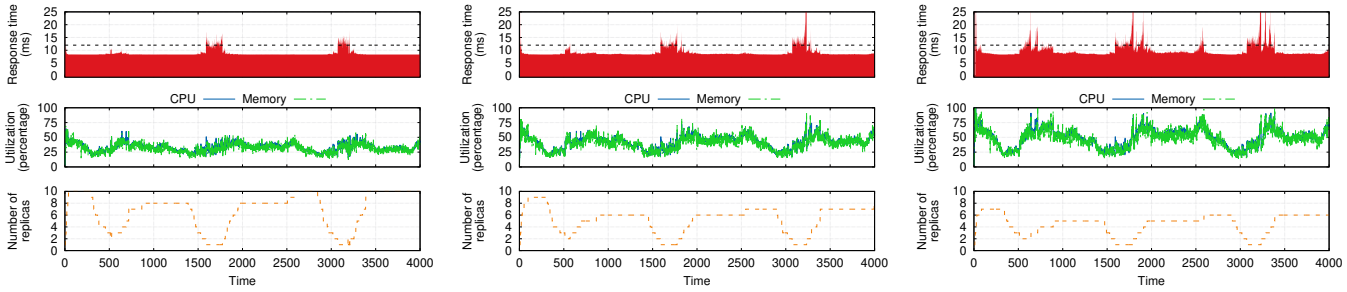
utilization, whereas the scale-out threshold can range between $_{\min} = 50\%$ and $_{\max} = 90\%$. The RL algorithms use the following parameters: $= 0{:}1$ (QL Threshold), $= 0{:}1$ (MB Threshold), and discount factor $= 0{:}99$. The discount factor , being close to 1, allows the RL agent to act with foresight, while the smoothing factors and enable to adapt the learned policies to (possibly) varying system dynamics. We discretize the application state with $u = r = 0{:}1$, and we set the threshold adaptation quantum to $= 5\%$. The cost function parameters are $= 10$ and $= \frac{}{_{\max}}$; they result from preliminary evaluations, where we wanted the two cost contributions to range in the same interval. To update the threshold, QL and DQL Threshold use an -greedy action selection policy, with $= 0{:}1$. DQL uses a replay memory with capacity of 50 observations and a batch size of 30; the target Q-network update frequency is $= 5$ time units. Configuring correctly the Q-network is an empirical task, which required some effort and several preliminary evaluations [29]. We implement the neural network using Deeplearning4j [43]. In particular, we use the rectified linear function as neuron activation function: due to its non-linear behavior, it is one of the most commonly used functions. To initialize the Q-network weights, we use the Xavier method [44]. To avoid weights to diminish or explode during network propagation, this method scales the weight distribution on a layer-by-layer basis. As regard the Q-network architecture, we select two distinct configurations for the single-agent and the multi-agent. The single-agent DQL Threshold uses a Q-network architecture that is fully-connected with 5 layers with $f4; 12; 12; 15; 3g$ neurons. The multi-agent uses a neural network with 4 layers with $f4; 12; 12; 3g$ neurons.

5.5 Optimization Objectives

This first set of experiments aims to show the flexibility of RL-based solutions to dynamically adapt the scaling thresholds. As shown in [6], a static threshold-based policy suffers from lack of flexibility and SLO unawareness, because it is not easy to satisfy a user-oriented SLO by setting thresholds on a system-oriented metric. This task is even more challenging when we consider multiple scaling metrics. Conversely, in the proposed approach, dynamic thresholds can be trained to optimize different deployment objectives, i.e., minimize $T_{\max}$ violations, the average number of replicas, or their combination. We consider that the application receives an incoming request rate that varies over time (see Fig. 2). It is a surrogate workload based on NYC Taxi Ride traces [45], where CPU and memory demand have a similar trend, so both can trigger scaling decisions.

TABLE 1
Application performance using different dynamic and multi-metric threshold-based scaling policies. Single-agent (SA) and Multi-agent (MA) architectures are considered, with different model-based and model-free RL solutions for updating the scaling thresholds.

Architecture	*Policy*	*Configuration* $\langle w_{\text{perf}}; w_{\text{res}} \rangle$	*Average CPU threshold (%)*	*Average CPU utilization (%)*	*Average Memory threshold (%)*	*Average Memory utilization (%)*	*Median response time (ms)*	$T_{\max}$ *violations (%)*	*Memory violations (%)*	*Average number of replicas*
SA	**MB Threshold**	$\langle 1; 0 \rangle$	50.52	33.56	50.18	32.02	8.35	5.42	0	6.85
		$\langle 0{:}5; 0{:}5 \rangle$	68.60	40.07	68.71	38.25	8.43	6.60	0	5.68
		$\langle 0; 1 \rangle$	89.99	50.32	89.98	48.02	9.00	14.07	0.07	4.48
	QL Threshold	$\langle 1; 0 \rangle$	63.75	34.85	67.43	33.27	8.36	6.22	0	6.63
		$\langle 0{:}5; 0{:}5 \rangle$	70.71	36.59	71.70	34.93	8.37	6.15	0	6.25
		$\langle 0; 1 \rangle$	75.63	38.81	78.58	37.05	8.42	6.65	0	5.84
	DQL Threshold	$\langle 1; 0 \rangle$	64.50	35.09	52.94	33.47	8.35	6.17	0.02	6.62
		$\langle 0{:}5; 0{:}5 \rangle$	86.36	41.49	86.30	39.59	8.41	10.70	0.05	5.64
		$\langle 0; 1 \rangle$	85.98	48.14	87.15	45.95	8.73	12.90	0.05	4.71
	DQL Threshold (pre-trained)	$\langle 1; 0 \rangle$	64.04	33.71	51.08	32.17	8.35	5.50	0	6.83
		$\langle 0{:}5; 0{:}5 \rangle$	87.99	42.69	86.67	40.76	8.50	9.65	0.05	5.49
		$\langle 0; 1 \rangle$	88.71	49.75	87.45	47.47	9.00	13.32	0.07	4.55
MA	**MB Threshold**	$\langle 1; 0 \rangle$	50.10	33.58	50.01	32.04	8.35	5.67	0	6.85
		$\langle 0{:}5; 0{:}5 \rangle$	67.23	42.43	73.90	40.50	8.54	7.22	0	5.31
		$\langle 0; 1 \rangle$	89.99	50.32	89.99	48.02	9.00	14.07	0.07	4.48
	QL Threshold	$\langle 1; 0 \rangle$	70.38	35.43	68.52	33.82	8.36	6.05	0	6.50
		$\langle 0{:}5; 0{:}5 \rangle$	75.56	40.76	82.84	38.92	8.46	7.42	0	5.58
		$\langle 0; 1 \rangle$	86.82	48.03	87.48	45.85	8.81	10.72	0	4.70
	DQL Threshold	$\langle 1; 0 \rangle$	52.50	33.68	52.64	32.14	8.35	5.45	0	6.82
		$\langle 0{:}5; 0{:}5 \rangle$	67.24	39.18	66.56	37.39	8.42	6.70	0	5.80
		$\langle 0; 1 \rangle$	73.71	41.62	71.26	39.72	8.51	7.75	0	5.49
	DQL Threshold (pre-trained)	$\langle 1; 0 \rangle$	52.20	33.67	52.62	32.12	8.35	5.55	0	6.83
		$\langle 0{:}5; 0{:}5 \rangle$	65.07	38.00	80.03	36.29	8.41	6.95	0	6.01
		$\langle 0; 1 \rangle$	81.92	43.52	78.42	41.55	8.53	10.05	0.02	5.34

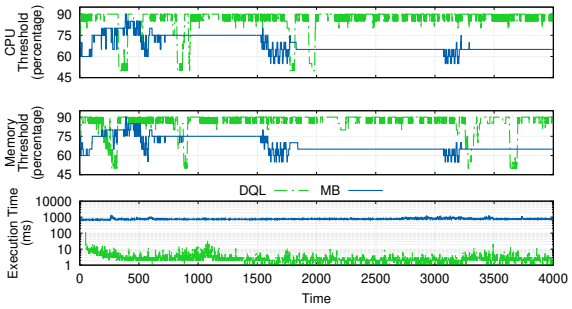


(a) Weights: $w_{\text{res}} = 0$ and $w_{\text{perf}} = 1$. (b) Weights: $w_{\text{res}} = 0{:}5$ and $w_{\text{perf}} = 0{:}5$. (c) Weights: $w_{\text{res}} = 1$ and $w_{\text{perf}} = 0$.

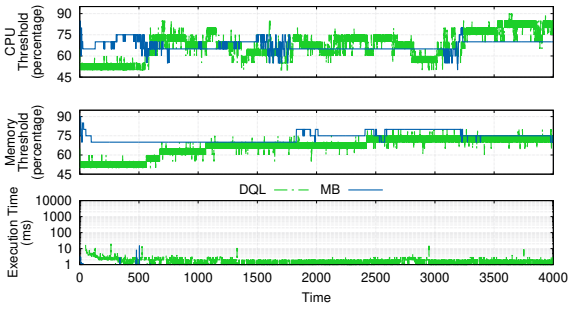
Fig. 3. Application performance using Multi-agent MB Threshold under different configurations of the immediate cost function.

We consider the SA and MA approaches, each with QL Threshold, MB Threshold, and DQL Threshold. Table 1 summarizes the experimental results. First, we discuss the results obtained with the MA approach. Overall, we can see that the RL-based approaches are flexible and can be tuned to optimize different deployment objectives, as follows from (1). Using an estimated model of the system dynamic, MB Threshold can successfully learn a different scaling threshold update strategy according to the different weight configurations. The effect of the different weights clearly appears by comparing the application performance in Fig. 3. When the cost function penalizes response time violations (i.e., with $w_{\text{perf}} = 1$), the average threshold values are rather close to 50%, which implies more frequent scale-up adaptations and thus a relative high number of replicas with an average number of 6:85 (see Fig. 3a). Since the application is readily scaled, the resulting response time is below $T_{\max}$ most of the time. When we aim to save resources (i.e., $w_{\text{res}} = 1$), the microservice $T_{\max}$ violations grow to 14:07% and the microservice response time registers different response time peaks when the CPU utilization is approaching 75% (see Fig. 3c). In this case, we obtain an average threshold value close to 90%, less frequent scale-ups and the resulting average number of replicas decreases to 4:48, corresponding to a 35% reduction of resources consumption. By varying the weights we obtain a wide set of adaption strategies. With equal weights, that is, $w_{\text{perf}} = w_{\text{res}} = 0{:}5$, we obtain an average CPU and memory threshold value of 67% and 74%, respectively. From Fig. 3b, we can observe that such behavior is needed to meet $T_{\max}$ requirements and avoid resource wastage.

We now compare MB Threshold against the model-free RL solutions, which obtain, in general, a worse application performance. From Table 1, we can see that QL Threshold frequently updates the scaling thresholds and only slightly differentiates the average threshold value for the various cost configurations. Differently from MB Threshold, it learns a less accurate application model, which results in sub-optimal thresholds for the cost function weights $w_{\text{perf}} = 1$. This behavior is also partially present in DQL Threshold; nevertheless, thanks to the replay buffer and the target network, the latter can more quickly converge to a stable solution. DQL Threshold learns good scaling thresholds for $w_{\text{perf}} = 1$ and $w_{\text{perf}} = w_{\text{res}} = 0{:}5$, resulting in $T_{\max}$ violations and average resource utilization close to those obtained by MB Threshold. When $w_{\text{res}} = 1$, DQL Threshold slowly learns the threshold update policy. This depends on the Q-function approximation by DQL; in the first half of the experiment, DQL Threshold progressively updates the threshold up to a value ranging between 80% and 90%. This setting results in a slightly higher microservice replication degree and reduced resource utilization with respect to MB Threshold. Although we could pre-train the Q-network to further improve DQL Threshold (mitigating



(a) Scale-out threshold values and Q-value update execution time when the SA architecture computes the CPU and memory scaling thresholds.



(b) Scale-out threshold values and Q-value update execution time under the MA architecture, where the *u*-agent and *r*-agent compute the scaling threshold on CPU and memory utilization, respectively.

Fig. 4. Threshold updates and execution time analysis of the different RL solutions, when the cost functions weights are $w_{res} = w_{perf} = 0.5$.

also the initial exploration phase), the obtained results are already remarkable, considering its model-free nature.

5.6 Comparing Single- and Multi-agent Architectures

The SA and MA architectures use a different model of the system state as well as of the available actions. The SA uses a system model with higher cardinality than MA, with 9801 state configurations and 5 actions instead of 99 and 3, respectively. Also in this case, QL Threshold is the approach that behaves worse than the others, because it needs a lot of samples before learning a good threshold adaptation strategy. During the experiment, it continuously explores the state-action pairs, learning a rather inaccurate system model. Differently from MA, in this case a larger number of configurations should be explored. When the SA uses DQL Threshold or MB Threshold, it learns different threshold update strategies according to the cost function weights, as in the previous section. So we need to delve into the key differences of the two approaches. In Fig. 4, we show the CPU and memory scaling threshold updates for MB Threshold and DQL Threshold as well as the time needed to run the RL agent and update the Q-function (referred to as execution time). We only show the more challenging case having $w_{perf} = w_{res} = 0.5$. Figure 4a shows that SA DQL Threshold learns scaling thresholds that tend to save resources instead of finding a trade-off with the application requirement satisfaction. This mainly depends on the problem space cardinality, which requires the Q-network to see a larger number of samples before correctly

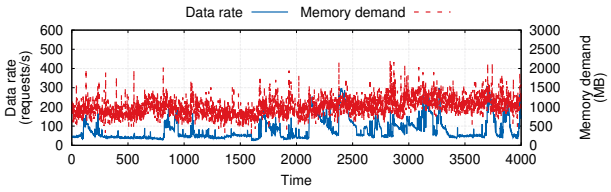


Fig. 5. Workload based on Bitbrains Rnd traces.

approximating the Q-function. DQL Threshold has a very limited execution time, which is always below 30 ms. Conversely, MB Threshold computes a better threshold update policy for both CPU and memory utilization. Nevertheless, the complexity of this approach clearly appears by looking at its execution time in Fig. 4a.

With the MA architecture, each RL agent can more easily learn a good adaptation policy, since it has to explore a reduced number of state-action pairs. As shown in Fig. 4b, DQL Threshold and MB Threshold set the threshold at around the 50% of the scale-out threshold range, identifying a good trade-off between T_{max} violations and average resource utilization. MB Threshold uses a more accurate system model, so it changes less often the thresholds (see Table 1). Figure 4b shows how the MA architecture drastically drops the execution time of MB Threshold, which is now even lower than the execution time by DQL Threshold.

5.7 Microservices with Different Critical Resource

The proposed scaling policies adapt the deployment of microservices having different type of critical resource.

5.7.1 Bitbrains Workload

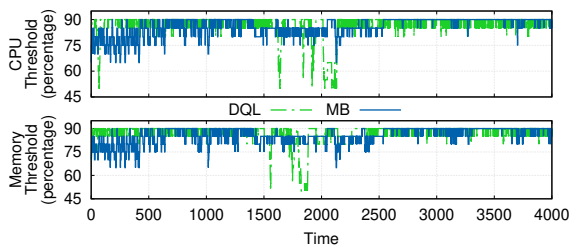
In this experiment, we benchmark the proposed dynamic threshold policies using the Rnd dataset from the GWA-T-12 Bitbrains workload trace [46]. Bitbrains is a service provider specialized in managed hosting and business computation for enterprises. The Rnd dataset consists of the resource usages of virtual machines used by the application services hosted within the Bitbrains data center. We accordingly amplified the dataset to further stress our microservices, obtaining the workload in Fig. 5. Differently from the previous experiments, in this case the application is mostly memory-demanding. Fig. 6 shows the threshold update during the experiment for the SA and MA architectures for the cost configuration $w_{res} = w_{perf} = 0.5$. Also in this case, MA performs better than SA, allowing the RL agent to more quickly learn the threshold update policy. We observe that MA leads to a lower scaling threshold on memory usage, because it learns that memory is the bottleneck resource. By setting the memory threshold on average on 68% and 67% respectively, MB and DQL Threshold lead to similar application performance, resulting in 5 microservice replicas and a 0.02% of T_{max} and memory demand violations.

5.7.2 CPU and Memory Intensive Workloads

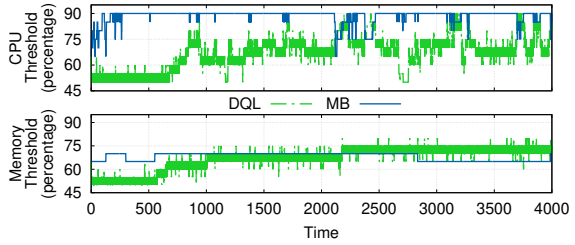
We now consider two types of synthetic workload, each with a different dominant resource request (see Fig. 7). Table 2 summarizes the experimental results. Also in this case, MA performs better that SA, and the MA with MB Threshold performs slightly better than DQL Threshold, allowing

TABLE 2
Application performance for the synthetic CPU-intensive and memory-intensive workloads, using MA with different scaling policies.

Workload	*Policy*	*Configuration* $\langle w_{perf}; w_{res} \rangle$	*Average CPU threshold (%)*	*Average CPU utilization (%)*	*Average Memory threshold (%)*	*Average Memory utilization (%)*	*Median response time (ms)*	T_{max} *violations (%)*	*Memory violations (%)*	*Average number of replicas*
CPU-intensive	**MB Threshold**	⟨1; 0⟩	50.01	36.20	50.10	8.69	8.43	1.42	0	5.71
		⟨0:5; 0:5⟩	67.94	42.46	89.99	10.19	8.74	2.12	0	4.93
		⟨0; 1⟩	89.99	56.26	89.99	13.50	10.75	35.17	0	3.75
	DQL Threshold	⟨1; 0⟩	52.32	36.20	52.59	8.69	8.43	1.42	0	5.71
		⟨0:5; 0:5⟩	64.66	39.23	66.12	9.41	8.47	2.87	0	5.42
		⟨0; 1⟩	73.41	45.77	74.14	10.98	8.74	12.60	0	4.64
Mem-intensive	**MB Threshold**	⟨1; 0⟩	50.01	9.03	50.01	32.50	8.33	0	0	5.04
		⟨0:5; 0:5⟩	87.55	11.18	86.56	40.24	8.34	0	0	4.03
		⟨0; 1⟩	89.99	15.17	89.99	54.61	8.36	0.07	0.17	3.26
	DQL Threshold	⟨1; 0⟩	52.54	9.03	52.66	32.50	8.33	0	0	5.04
		⟨0:5; 0:5⟩	64.35	10.43	67.24	37.56	8.33	0	0	4.34
		⟨0; 1⟩	71.82	11.23	73.65	40.43	8.34	0	0	4.03
	u-agent **MB Threshold**	⟨1; 0⟩	50.01	25.95	-	93.43	9.29	10.07	46.01	1.81

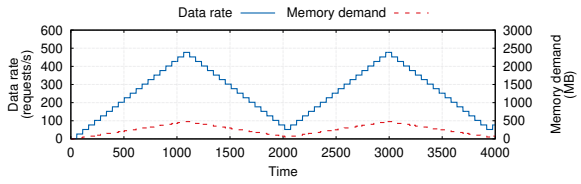


(a) Threshold values when the SA solution computes the CPU and memory scaling thresholds.

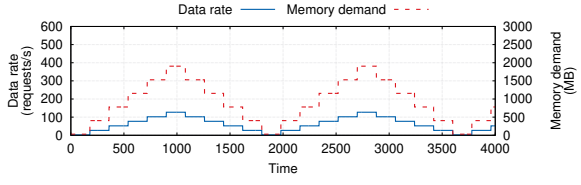


(b) Threshold values with the MA solution.

Fig. 6. Threshold updates for the Bitbrains Rnd workload, when the cost function weights are $w_{res} = w_{perf} = 0.5$.



(a) CPU-intensive workload.



(b) Memory-intensive workload.

Fig. 7. Synthetic workloads with different pattern of resource request.

the RL agent to more quickly learn a suitable threshold adaptation policy. We first discuss the CPU intensive workload considering the most challenging cost configuration, i.e., $w_{res} = w_{perf} = 0.5$ (Fig. 7a). MB Threshold quickly identifies the CPU as the critical resource, so it sets a lower scaling threshold for CPU than for memory (respectively 67:94% and 89:99%, on average). The memory is not the bottleneck resource and, although it would encourage scale-in actions, the CPU utilization actually drives the microservice adaptation. We observe a similar performance also for DQL Threshold. On average, DQL Threshold results in a slightly higher number of microservice replicas than MB Threshold, due to the different memory threshold average value. Moreover, it also changes the thresholds more frequently.

When we consider the memory intensive workload (Fig. 7b), we expect that a scaling policy that only takes into account CPU resources would be unable to successfully adapt the application deployment. As an example, we run the MA with MB Threshold by disabling the *r*-agent (i.e., with only the *u*-agent that considers CPU) and by setting $w_{perf} = 1$ in the cost function. Although the agent uses a scale-out threshold on 50% of CPU utilization, scaling actions are rarely triggered, resulting in 46% memory demand violations. Conversely, the proposed multi-metric solutions (i.e., MB and DQL Threshold) can correctly react to CPU and memory demand changes. When $w_{perf} = 1$, both MB and DQL Threshold result in no memory violations using, on average, 5:04 microservice replicas. Also for this workload, the MB approach succeeds in improving the threshold update strategy, especially when the cost function requires to save resources (i.e., $w_{res} = 1$).

5.8 Discussion

We extensively evaluated the proposed RL-based threshold update strategies using two different architectures: SA and MA. First, we showed the flexibility provided by a RL-based solution for updating the scaling thresholds. By correctly defining the relative importance of the deployment objectives through the cost function weights in (1), the RL-agent can accordingly learn a suitable threshold update strategy. Second, we showed that a model-based RL approach takes advantage of the system model to estimate the effect of performing an action in a given state. Although this allows to boost the learning phase, defining the model is not a trivial task and, when multiple metrics are considered, it can penalize the execution time. To overcome this issue, we proposed the MA architecture, which shows good performance at a reduced Q-function update execution time (due to the smaller state-action pairs cardinality). Our experiments showed that MA with MB Threshold is the best strategy for controlling dynamic multi-metric thresholds. Since determining the system model can be challenging, we also explored a DQL-based approach. Although we do

not need to define the system model, DQL introduces the effort of defining the Q-network architecture, which is an empirical process and may require a large number of trial-and-error repetitions. Note that we adopted two different network architectures for SA and MA. We showed that DQL Threshold outperforms QL Threshold and achieves performance close to MB Threshold, especially when we pre-train the Q-network. To conclude, we showed that the proposed approaches correctly perform under different application workloads (i.e., CPU-intensive, memory-intensive, mixed).

6 Prototype-based Experiments

We evaluate the proposed scaling policies in a real environment. First, we present the basic concepts of Kubernetes and of the custom scaling policies integration into it. Second, we present three state-of-the-art scaling policies against which we compare our own. Then, we evaluate our policies with CPU- and memory-intensive microservices.

6.1 Integration into Kubernetes

Kubernetes is an open-source orchestration platform that simplifies the deployment, management, and execution of containerized applications. A pod is the smallest deployment unit in Kubernetes. It consists of one or more tightly coupled containers that are co-located and scaled as an atomic entity. Each application microservice is deployed using a pod. Kubernetes allows to configure each pod with specific resource requests and limits. A resource request is the minimum amount of (CPU and memory) resources needed by the pod. A resource limit is the maximum amount of resources that can be assigned to the pod.

To integrate new deployment policies in Kubernetes, we provide a custom auto-scaler that follows a MAPE control loop [47]. At each loop iteration, our auto-scaler monitors the environment and the controlled microservice; then, it analyzes application-level and system-level metrics; accordingly, it plans the scaling actions which are then executed using the Kubernetes APIs. The different scaling policies implement the planning phase of the MAPE control loop. Updating the thresholds does not restart containers. Then, the auto-scaler adds or removes containers based on CPU and memory usage by the service. Scaling-out operations do not affect the microservice's instances currently running.

6.2 Benchmark Policies

Horizontal Pod Autoscaler. Kubernetes includes the Horizontal Pod Autoscaler (HPA), which relies on CPU utilization to horizontally scale a single microservice deployment [48]. HPA monitors the CPU utilization of the microservice pods. It scales the number of pods according to the ratio between the observed value and the target value of CPU utilization. Recently, Kubernetes provides a beta API to autoscale deployment on multiple observed metrics (e.g., CPU and memory utilization). After evaluating each metric individually, HPA uses the most critical one to take scaling decisions. HPA relies only on static thresholds.

HyScale. HyScale [7] combines horizontal and vertical scaling to adapt the microservice deployment. It adjusts the CPU and memory limit of each pod, aiming to make resource utilization as close as possible to the target one. HyScale gives priority to vertical scaling and performs horizontal scaling only if the required amount of resources cannot be acquired otherwise. HyScale plans scaling actions by using target utilization values (similarly to HPA).

Horovitz et al. Horovitz et al. [8] propose a model-free Q-learning approach to dynamically adapt the CPU scaling thresholds. It uses the number of microservice replicas as state and the threshold update as action. An additional data structure is required for mapping each state to the last exploited threshold value. Differently from our approach, the proposed solution also uses an additional heuristic to determine whether to update and activate the RL agent.

6.3 Experimental Setup

We run the experiments on a cluster of 5 virtual machines of the Google Cloud Platform; each virtual machine has 2 vCPUs and 4 GB of RAM (type: e2-medium).

To evaluate our dynamic threshold scaling policies, we consider two microservices: *Pi* and *Word-count*. They are RESTful web services, implemented in Python using Flask. Pi is a CPU-intensive service: upon request, a Monte Carlo algorithm approximates after placing 1000 random points in a unit square. Word-count is a memory-intensive microservice: for each sentence received as request, the service extracts its words and returns the updated word count. The counter keeps track of the word occurrences received in the last seconds. Each word-count pair is kept in memory. We deploy each service instance using a pod with 500 millicore (i.e., 0.5 vCPU) and 256 MB of RAM.

We parametrize the RL-based policies as in Section 5.4. The Pi service requires its response time to be below $T_{\max} = 150$ ms and sets the scale-in threshold to 20% of CPU and memory utilization. Word-count requires $T_{\max} = 100$ ms and we use a count window of $= 120$ s. For Word-count, we set the scale-in thresholds of CPU and memory utilization to 20% and 35%, respectively. The latter threshold accounts for the service's baseline memory footprint. We compute the application response time as the average request completion time over a tumbling window of 10 s.

6.4 Scaling Policies Evaluation

In this section, we show the benefits of multiple scaling thresholds when heterogeneous microservices are deployed using Kubernetes. We do not present QL Threshold due to space limits; however, it does not improve its performance over the other solutions (see Section 5.3). For the RL-based solutions, we consider the most challenging set of weights $w_{\text{perf}} = w_{\text{res}} = 0.5$. In HyScale, the CPU and memory limit ranges in the interval $[0.25; 1]$ vCPU and $[100; 1024]$ MB, respectively. Table 3 summarizes the experimental results.

CPU-intensive Microservice: Pi. The microservice receives a workload that follows the NYC Taxi Ride trace [45], accordingly amplified and accelerated so to further stress the service resource requirements, as shown in Fig. 8. When the SA architecture is considered, MB Threshold deploys on average 2.26 instances, setting the CPU scaling threshold to 75.96%. Being Pi a CPU-intensive microservice, we register a memory utilization that is, on average, below the scale-in threshold. As a consequence, the CPU utilization actually

TABLE 3
Prototype-based experiments: Application performance with different threshold-based scaling policies and applications.

Application	Architecture	*Policy*	*Average CPU threshold (%)*	*Average CPU utilization (%)*	*Average Memory threshold (%)*	*Average Memory utilization (%)*	*Average Share CPU/Mem (MB)*	*Median response time (ms)*	*T_{max} violations (%)*	*Memory violations (%)*	*Average number of replicas*
Pi	SA	**MB Threshold**	75.96	43.35	76.00	15.59		90.38	6.92	0	2.26
		DQL Threshold	83.05	48.93	67.92	18.43		75.24	8.81	0	1.70
		DQL Threshold (pre-trained)	89.09	49.30	88.80	15.69		85.14	9.43	0	1.74
	MA	**MB Threshold**	79.27	47.66	89.94	15.25		73.66	8.99	0	1.98
		DQL Threshold	52.58	34.30	52.75	14.77		66.81	1.12	0	2.79
		DQL Threshold (pre-trained)	72.78	42.29	82.61	15.09		76.47	5.56	0	2.12
	-	**HPA 80/80**	80.00	49.94	80.00	15.48		82.12	9.59	0	1.56
		HPA 70/70	70.00	47.75	70.00	17.53		72.57	6.03	0	1.61
		HPA 60/60	60.00	43.37	60.00	14.87		70.05	5.46	0	1.70
	-	**HyScale 80/80**	80.00	63.09	80.00	37.27	0.388/100	115.28	47.56	0	1.00
		HyScale 70/70	70.00	58.46	70.00	35.81	0.440/100	82.86	35.00	0	1.02
		HyScale 60/60	60.00	53.49	60.00	36.15	0.511/100	72.42	26.76	0	1.11
	-	**Horovitz et al.**	57.20	34.21	-	15.99		12.19	1.51	0	2.68
Word-count	SA	**MB Threshold**	78.83	26.64	78.71	52.93		6.56	5.65	0	2.69
		DQL Threshold	76.35	33.88	84.29	50.83		8.02	14.18	0.71	2.52
		DQL Threshold (pre-trained)	88.25	42.13	88.59	59.20		17.11	21.25	0.63	1.64
	MA	**MB Threshold**	78.06	27.96	73.41	53.51		6.35	8.24	0	2.56
		DQL Threshold	52.38	21.94	52.86	44.54		6.29	4.76	0	3.40
		DQL Threshold (pre-trained)	68.68	31.08	71.67	53.13		7.68	11.49	0	2.31
	-	**HPA 80/80**	80.00	47.01	80.00	53.03		30.47	30.29	0	1.20
	-	**HPA 70/70**	70.00	28.04	70.00	55.58		6.81	4.29	0	2.23
	-	**HPA 60/60**	60.00	25.33	60.00	47.14		5.91	1.42	0	2.61
	-	**HyScale 80/80**	80.00	51.51	80.00	40.97	0.303/200	35.95	29.07	0	1.00
		HyScale 70/70	70.00	49.96	70.00	39.59	0.352/201	32.88	25.00	0	1.00
		HyScale 60/60	60.00	46.16	60.00	37.03	0.314/200	42.26	32.56	0	1.00
	-	**Horovitz et al.**	58.68	34.34	-	59.17		8.67	9.37	1.56	1.72

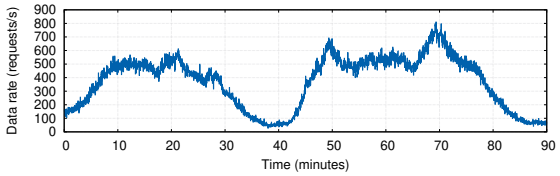
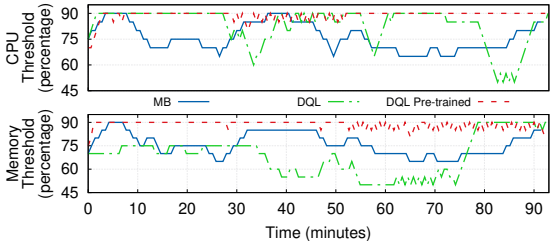
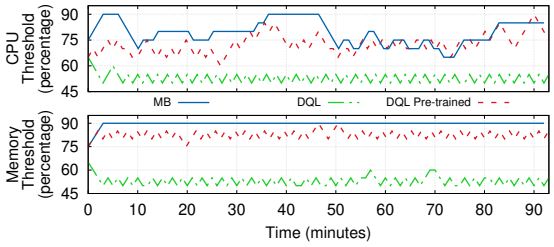


Fig. 8. Workload used for the Pi service.



(a) Scale-out threshold values with a single-agent solution.



(b) Scale-out threshold values with the multi-agent solution.

Fig. 9. Threshold updates for the Pi service, when the cost function weights are $w_{perf} = w_{res} = 0.50$.

drives the microservice adaptation although the memory encourages scale-in actions. When DQL Threshold is considered, we observe a slightly worse performance. DQL Threshold learns an adaptation policy that prefers to save resources instead of finding a trade-off between application performance and resource utilization. It uses on average 1.70 replicas, registering 8.81% of T_{max} violations. To mitigates the initial exploration of the *tabula rasa* approaches (as by MB and DQL Threshold solutions), we could pre-train the Q-network of DQL. However, we do not observe significant performance improvement. This happens because of the complexity of the SA architecture combined with the model-free nature of DQL. We now consider the MA architecture. As shown in Fig. 9, the MB approach succeeds identifying the CPU as the bottleneck resource. So, it sets a lower scaling threshold for CPU than for memory. MB Threshold identifies an adaptation policy that runs the microservice using, on average, 1.98 instances with an average CPU utilization of 47.66%. Conversely, DQL Threshold is not able to find a suitable trade-off between performance degradation and resource wastage at run-time. Setting on average the thresholds to 53%, DQL Threshold deploys a higher number of deployed replicas with a low 34.30% of CPU utilization. We need to pre-train the Q-network to improve the performance of DQL Threshold (see Table 3).

We now compare our results against the policies described in Section 6.2. HPA and HyScale are application-unaware and require to manually set the thresholds on CPU and memory utilization. In HPA, changing the CPU scaling threshold affects the microservice performance: the number of T_{max} violations decreases from 9.59% to 5.46% when setting the threshold from 80% to 60%; the average number of microservices replicas increases from 1.56 to 1.70. Differently from the previous policies, HPA does not immediately react to load variations, but allows a limited time interval before performing the scaling. HPA uses a simple strategy, which however moves the complexity on the threshold definition. It can obtain valid scaling strategies, but we need to manually tuning its parameters, and explore the effect of the different thresholds (see Table 3). HyScale extends HPA preferring vertical to horizontal scaling. HyScale sets the memory share to 100 MB, the minimum value for memory limit, which is reasonable being Pi CPU-intensive. When the CPU scaling threshold is increased from 60% to 80%, the average CPU utilization increases from 53.49% to 63.09%, which is obtained by decreasing the average CPU share of pods from 510 to 388 millicore. The other policies can only assign multiple of 500 millicore. Although improving uti-

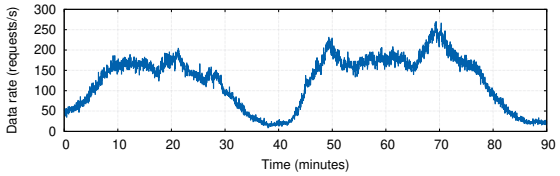


Fig. 10. Workload used for the Word-count service.

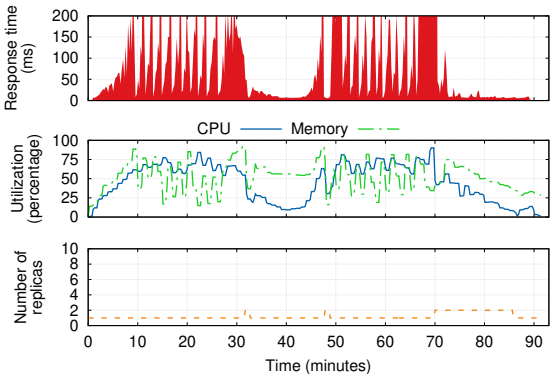
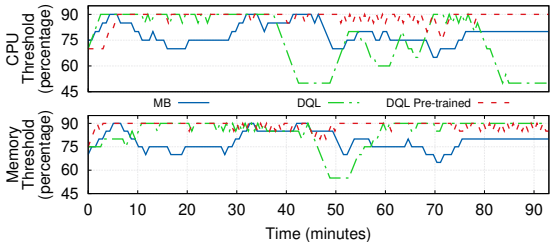
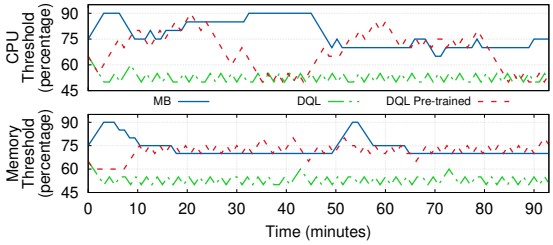


Fig. 11. Word-count: Service performance using Kubernetes HPA and setting to 80% the memory and CPU scaling thresholds.



(a) Threshold value with a single-agent solution.



(b) Threshold value with the multi-agent solution.

Fig. 12. Threshold updates for the Word-count service when the cost function weights are $w_{\text{perf}} = w_{\text{res}} = 0{:}50$.

lization is desirable, HyScale results in a very high number of T_{max} violations (even with 60% as CPU threshold). This is also due to the high number of vertical scaling operations performed (63% of the time). To implement vertical scaling, Kubernetes gradually creates pods with the new configuration and deletes the old ones; in this stage, the application availability decreases. Differently from HPA and HyScale, our policies can automatically learn how to set and update the scaling thresholds, according to the desirable deployment goals. Dynamic thresholds have been proposed also by Horovitz et al.; their solution uses Q-leaning to update the CPU scaling threshold and avoid T_{max} violations. Table 3 shows that this policy meets the target response time at the cost of a very low average CPU utilization (on average, 34:21%) and a high number of pods (on average, 2:68). This policy computes a very low CPU scaling threshold (57:20, on average), which mainly results from the update heuristic that activated the RL agent only 7:58% of the time during the experiment. Conversely, our policy can be tuned, as in this experiment, so to limit response time violations while avoiding resource under-utilization. Importantly, we do not define custom update rules, but we rely on pure learning approaches, where the agent itself learns how to configure the thresholds by exploiting experience and an approximated system model.

Memory-intensive Microservice: Word-count. The workload's request pattern for the Word-count microservice is shown in Fig. 10. From Table 3, we observe that higher values of memory utilization emphasize the memory-intensive nature of this service. Note also that, although the threshold values are, on average, quite similar, the monitored CPU utilization is, on average, significantly lower than the threshold. Hence, for Word-count it is the memory utilization that drives the deployment adaptation by triggering scale-out actions. When the SA architecture is adopted, MB Threshold identifies an adaptation policy that runs the microservice using, on average, 2:36 replicas and achieves an average CPU and memory utilization of 30:40% and 53:94%, respectively. Fig. 12a shows that DQL Threshold cannot efficiently update the scale-out thresholds, whose values range from 50% to 90%. Pre-training the Q-Network does not lead to a significant improvement, due to the challenging learning task required by the SA architecture. Conversely, the MA architecture simplifies the learning task, resorting to two RL agents (i.e., u-agent and r-agent) that operate on a reduced size space. This leads to an overall performance improvement. Differently from Pi, Word-count requires to carefully monitor both CPU and memory (compare Fig. 12 against Fig. 9): each request is served by a service thread leading to both memory and CPU usage increment as the number of requests increases as well. As shown in Table 3, also in this case, MB Threshold learns a better adaptation strategy. On average, it sets the memory and CPU scaling thresholds to 77:15% and 78:02%, respectively. DQL Threshold slowly learns how to adapt the scaling threshold, resulting on an average threshold value of 53% for both CPU and memory. This results in a higher number of service replicas (on average, 3:40 pods are deployed instead of 2:56). In this case, pre-training the Q-network clearly improves the DQL Threshold behavior, which can quickly identify the memory as the bottleneck resource (see Fig. 12b). Determining static thresholds for Word-count using HPA is more challenging than for Pi. Table 3 shows that small changes on the scale-out thresholds lead to very different application performance. When HPA uses the CPU and memory scale-out thresholds set to 80%, it deploys 1:20 replicas on average and registers a CPU and memory utilization of 47% and 53%, respectively. However, as shown in Fig. 11, the memory is often over-utilized and, as a consequence, the service is continuously restarted due to out-of-memory exceptions. By setting the thresholds to 70% or 60%, the application achieves better performance; nonetheless, we might still perform a finer

thresholds' tuning to find an application deployment that better satisfies our goals. HyScale achieves performance results worse than the other policies. For all the threshold values, it results in a very high number of T_{max} violations (about 30%), with an average CPU and memory utilization of about 50% and 40%, respectively. Also in this case, HyScale aggressively changes the microservice deployment most of the time through vertical scalings. The policy by Horovitz et al. does not consider memory utilization; as expected, since Word-count is memory-intensive, scaling it using only CPU utilization as metric leads to the highest value of memory demand violations. As for Pi, this heuristic updates the Q-table less than 13% of the time.

6.5 Discussion

The prototype-based experiments show the benefits of dynamic thresholds as well as of the MA architecture. Differently from HPA and HyScale, the proposed dynamic threshold-based policies avoid the process of manually tuning the scaling threshold, which may require a detailed profiling of the microservices to deploy. Moreover, the RL-based solutions allow to specify *what* to obtain instead of *how* to obtain it. Specifically, they allow to specify deployment goals in terms of user-oriented application metrics (e.g., response time), instead of system-oriented metrics. Exploiting an approximate system model, our RL solutions do not require additional rules or heuristics to boost learning (as Horovitz et al. do). The experiments confirm the positive impact of a MA architecture on RL agent scalability, thus reducing the execution time. Indeed, the average execution time of MB Threshold, on all the prototype-based experiments, dropped from 864 ms (SA architecture) to 1 ms (MA architecture). As regards the comparison between MB Threshold and DQL Threshold, the same conclusion of Section 5.3 holds: MB Threshold is more promising, because it benefits from the system model. However, also DQL Threshold achieves satisfactory results, especially if we resort on pre-training. This approach could be successfully adopted in contexts where modeling the system is hard.

7 Conclusion

Today's cloud providers support the elasticity of microservice-based applications by creating multiple, decentralized auto-scaler instances, each one in charge of adapting a single microservice. Threshold-based policies are the most popular strategy to efficiently scale microservices at run-time. Most cloud-native applications are heterogeneous in nature, so considering only a single metric to scale microservices turns out to be ineffective. Defining multiple thresholds manually to address different application needs is an error-prone and cumbersome task. Therefore, we proposed self-adaptive and multi-metric threshold-based policies to efficiently control the elasticity of microservices with different resource demands. We relied on RL to dynamically update the scaling thresholds for each relevant metric and, in particular, we designed different model-free and model-based approaches: QL Threshold, DQL Threshold, and MB Threshold. To reduce the control policy execution time when multiple metrics are considered, we also presented a MA architecture, that defines a RL agent for each controlled metric, and compared it against the more intuitive SA architecture. We conducted a thoroughly evaluation, showing the benefits of self-adaptive multi-metric thresholds and of the proposed RL-based approaches. MB Threshold exploits the estimated system dynamics to speed up the learning phase, leading to better application performance; however, modeling the system can be challenging. Therefore, we also explored a DQL-based approach, where a deep neural network learns to estimate the system dynamics.

As future work, we plan to consider additional resource metrics (e.g., network and I/O usage) and to devise global policies to seamlessly coordinate the scaling of multi-component applications.

Acknowledgments

We gratefully acknowledge the support received from Google with the GCP research credits program.

References

[1] V. Cardellini, F. Lo Presti, M. Nardelli, and F. Rossi, "Self-adaptive container deployment in the fog: A survey," in *Algorithmic Aspects of Cloud Computing*, ser. LNCS, vol. 12041. Springer, 2020.

[2] M. Saif, S. Niranjan, and H. Al-ariki, "Efficient autonomic and elastic resource management techniques in cloud environment: Taxonomy and analysis," *Wireless Netw.*, vol. 27, 2021.

[3] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu *et al.*, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. of ACM SoCC '21*, 2021, pp. 412–426.

[4] Google. (2021) Online boutique. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[5] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. of IEEE CLOUD '19*, 2019, pp. 329–338.

[6] F. Rossi, V. Cardellini, and F. Lo Presti, "Self-adaptive threshold-based policy for microservices elasticity," in *Proc. of IEEE MASCOTS '20*, 2020, pp. 1–8.

[7] A. Kwan, J. Wong, H. Jacobsen, and V. Muthusamy, "HyScale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. of IEEE ICDCS '19*, 2019, pp. 80–90.

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

[9] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Trans. Cloud Comput.*, vol. 8, no. 2, pp. 635–646, 2020.

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

[11] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *Proc. of IEEE IC2E '19*, 2019, pp. 200–210.

[12] Y. Guo, A. L. Stolyar, and A. Walid, "Online VM auto-scaling algorithms for application hosting in a cloud," *IEEE Trans. Cloud Comput.*, vol. 8, no. 3, pp. 889–898, 2020.

[13] S. Shevtsov, D. Weyns, and M. Maggio, "Self-adaptation of software using automatically generated control-theoretical solutions," in *Engineering Adaptive Software Systems*. Springer, 2019.

[14] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proc. of ACM SIGSOFT FSE '16*, 2016, pp. 217–228.

[15] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," *IEEE Trans. Serv. Comput.*, vol. 5, no. 4, pp. 497–511, 2012.

[16] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proc. of IEEE NOMS'12*, 2012, pp. 204–212.

[17] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *Proc. of IEEE ICDCS '19*, 2019.

[18] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. of IEEE ICDCS '19*, 2019, pp. 2015–2025.