Published in: Proceedings of the 28th IEEE Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2020), Nice, France, November 17-19 2020.
DOI: https://doi.org/10.1109/MASCOTS50786.2020.9285951

Self-adaptive Threshold-based Policy for Microservices Elasticity

Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti
DICII, University of Rome Tor Vergata, Italy
{f.rossi,cardellini}@ing.uniroma2.it, lopresti@info.uniroma2.it

***Abstract*—The microservice architecture structures an application as a collection of loosely coupled and distributed services. Since application workloads usually change over time, the number of replicas per microservice should be accordingly scaled at run-time. The most widely adopted scaling policy relies on statically defined thresholds, expressed in terms of system-oriented metrics. This policy might not be well-suited to scale multi-component and latency-sensitive applications, which express requirements in terms of response time.**

In this paper, we present a two-layered hierarchical solution for controlling the elasticity of microservice-based applications. The higher-level controller estimates the microservice contribution to the application performance, and informs the lower-level components. The latter accordingly scale the single microservices using a dynamic threshold-based policy. So, we propose MB Threshold and QL Threshold, two policies that employ respectively model-based and model-free reinforcement learning approaches to learn threshold update strategies. These policies can compute different thresholds for the different application components, according to the desired deployment objectives. A wide set of simulation results shows the benefits and flexibility of the proposed solution, emphasizing the advantages of using dynamic thresholds over the most adopted policy that uses static thresholds.

***Index Terms*—Hierarchical Control, Elasticity, Self-adaptation, Microservice, Reinforcement Learning**

I. Introduction

To take advantage of cloud computing and to improve efficiency and scalability of applications, most of the IT companies (e.g., Amazon, Netflix, Spotify) are currently reshaping their applications from monolithic architectures to microservices. According to the microservices architectural style, an application can be split into many autonomous and decoupled services, each providing a specific functionality.

Exploiting elasticity, each microservice can be dynamically scaled, enabling to control the application deployment with a fine granularity and to reduce the scaling cost compared to conventional monolithic solutions. Besides controlling elasticity of single components, microservice applications require to efficiently coordinate the distributed scaling decisions so to properly process varying workloads and meet application-level Quality of Service (QoS) requirements. Although elasticity has been widely explored in the context of cloud computing [1], scaling microservice applications, where multiple components loosely cooperate and interact with one another, has only recently started to be investigated (e.g., [2]–[5]). In this setting, the complexity of managing microservice applications which include, among the others, the challenges posed by the need to map application into microservices' requirements as well as the dynamism of the execution environments, demand novel and autonomic solution to control microservices elasticity. Today's cloud providers that support multi-component applications (e.g., using containers and container orchestration engines) allow to create multiple, decentralized auto-scaler instances, each carrying out the adaptation of a single microservice deployment. Policies to coordinate the scaling decisions at the application level are missing. To determine the scaling actions, most of existing auto-scalers use static thresholds on system-oriented metrics (e.g., CPU utilization)[1]. The main idea is to increase (or reduce) the microservices' parallelism degree as soon as the metric is above (or below) a critical scale-out (or scale-in) value. Although this decentralized approach scales well, we observe that manually tuning such scaling thresholds is challenging, especially when we need to define multiple thresholds, one for each microservice. A further challenge arises from the need of specifying a critical value on a system-oriented metric, whereas the application usually exposes its requirements in terms of user-oriented metrics (e.g., response time, throughput, cost).

In this paper, we design a self-adaptive threshold-based scaling policy that can automatically learn and update the scaling thresholds for each application component. Although self-adaptive thresholds have been already studied in different contexts (e.g., virtual machine consolidation [6], performance management [7]), to the best of our knowledge, they have never been applied to microservice applications. The main contributions of the paper are as follows.

- We propose a two-layered hierarchical solution for controlling elasticity. Having a complete system view, a high-level centralized entity estimates at run-time the relationship between application- and microservice-level QoS requirements (global policy). At low level, decentralized entities locally control the adaptation of single microservices (local policy).
- As local policy, we propose *QL Threshold* and *MB Threshold*, two dynamic threshold-based policies realized using model-free and model-based Reinforcement Learning (RL) solutions, respectively. Intuitively, a model-based approach allows us to account for the known (or estimated) system dynamic improving the algorithm learning speed. As global policy, we propose a simple heuristic

[1]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

that dynamically estimates and adapts each microservice contribution to the overall application performance.

- Using simulation, we demonstrate the benefits of a hierarchical control, the advantages of the proposed MB Threshold policy, and the flexibility of RL-based solutions that can identify different trade-offs between improving application performance and avoiding resource wastage. We also compare our solution against the widely adopted static threshold-based policy.

II. Related Work

In this section, we analyze existing approaches proposed in literature for the adaptive deployment of microservice-based applications in cloud environment. We broaden the view also to monolithic applications because, so far, only few research works [2]–[5] have specifically targeted the elasticity of microservice-based applications . Existing elasticity policies range from model-free to model-based solutions, according to the degree of system knowledge exploited to approximate the application behavior. A model-free solution requires no knowledge of the system dynamics; so it can take sub-optimal decisions or may require manual parameter tuning. Different model-based techniques have been proposed for application scaling, such as control theory [8], queuing theory [3], time series analysis [9], or a combination thereof [10]. The model-based approaches usually need many training samples that can be extract from historical data. In general, it is hard to perform off-line model training of microservice-based applications because they can be complex and have highly dynamic behaviors.

The main methodologies to adapt the application deployment are: threshold-based heuristics, fuzzy logic, queuing theory, and machine learning-based solutions. The most popular approach uses best-effort threshold to change the application replication degree at run-time (e.g., [7], [10]). Most works propose a static threshold-based approach and use, as QoS metric, the resource utilization of either the system nodes or the application replicas (e.g., [4], [10], [11]). Although threshold-based policies are simple to design, they require a manual threshold tuning that, in general, is not a trivial task. To overcome this issue, self-adaptive (or dynamic) threshold-based policies have been proposed in literature. They have been used in different contexts, such as performance management, virtual machines consolidation, scaling of monolithic applications (e.g., [6], [7], [12]); however, to the best of our knowledge, they have never been applied in a microservices-based scenario. Fuzzy logic approaches use pre-defined collections of if-then rules that represent how to take decisions and control a system according to the human knowledge (e.g., [13], [14]). When the scale of the controlled system increases, determining robust rules to combine conjunctive or disjunctive clauses becomes hard, also because we cannot give different importance to the factors to be combined. Queuing theory models an application as a queuing network. This allows to predict the application performance under different conditions of load and replication, and accordingly drive the scaling operations (e.g., [3], [10], [15]). Queuing models return only approximated behavior; moreover, parameterizing them correctly requires an extensive application profiling that can be time-consuming and costly. Recently, machine learning policies are becoming appealing to manage and adapt complex systems also in a fully decentralized manner (e.g., [2], [5], [16]). In this field, reinforcement learning is a special technique by which an agent can learn how to make good decisions through a sequence of interactions with the environment. Most of the works consider the model-free RL (e.g., Q-learning) algorithms (e.g., [7], [14], [17]) which, however, suffer from slow learning rate. As such, the auto-scaler performs poorly during the learning period. To overcome the slow convergence rate of these solutions, model-based RL approaches have been proposed. Tesauro et al. [18] use queing network to model the application performance. In [16], we present a model-based solution that empirically estimates the system model and scales monolithic applications.

In this paper, we resort to a threshold-based policy where we use RL to automatically learn and adapt the scaling thresholds at run-time. The work by Horovitz at al. [7] is the most closely related to ours. We differ from their solution from both the architectural and methodological standpoints. First, Horovitz et al. propose a centralized heuristic based on a model-free Q-learning approach to dynamically scale monolithic applications. Conversely, we design a two-layered hierarchical solution to adapt at run-time the deployment of multi-component applications. Exploiting a full system view, the high-level control entity estimates at run-time each microservice contribution to the application response time and accordingly notifies the low-level per-microservice managers. Moreover, we propose a model-based RL solution that can speed-up the learning phase of the RL agents by exploiting the knowledge on the system dynamics. As such, we do not rely on additional heuristics to determine whether to activate the RL agent (as done in [7]).

III. System Architecture

A. Problem Definition

A microservice-based application results by the cooperation of different independently deployable services. Nevertheless, the overall application performance results by the smooth integration and cooperation between its microservices. Without loss of generality, we can model a microservice-based application as directed acyclic graph (DAG), where the vertices represent the application microservices and the edges the logical links or dependencies between them [19].[2] Two services are interconnected if they directly communicate to reach a common goal (i.e., to satisfy an external request). Let M be the set of all the application microservices. We define the vertices without incoming (internal) links as *sources* and those without outgoing links as *sinks*. A front-end service (e.g., a gateway) is a source, because it can forward the user requests to the other microservices; a sink only returns a response, without

[2]The application DAG can be manually defined, e.g., in the deployment specification, or can be estimated at run-time using, e.g., a service mesh. Dependency cycles in microservice-based applications should be removed for data integrity and to reduce the risk of outages [20].

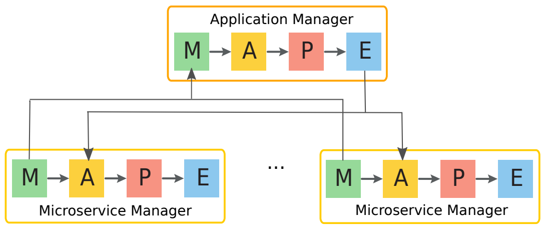


Fig. 1: High-level overview of the hierarchical control.

invoking other services. We define the set of all source-sink paths as Π. Note that a single microservice can be a member of multiple paths. Given a microservice $m \in M$, we denote the set of all paths that include m as $\Pi_m \subseteq \Pi$.

In this work, we consider latency-sensitive applications that expose QoS requirements in terms of a target response time that should not be exceeded (i.e., T_{max}). Since the application workload usually fluctuates over time, the number of replicas of each microservice should be accordingly scaled at run-time so to meet the T_{max} requirement avoiding resource wastage. Multiple microservice replicas can process incoming requests in parallel, thus reducing the per-replica load and, in turn, the processing latency.

B. Hierarchical Control Architecture

To manage and coordinate the microservices auto-scaling so to obtain desirable application performance, we need a deployment controller that provides self-adaptation mechanisms and can be equipped with deployment policies. The MAPE loop represents a prominent and well-know architectural pattern to organize the deployment controllers, where four components (Monitor, Analyze, Plan, and Execute) are responsible of self-adaptation actions. As described in [21], different patterns have been used in practice to decentralize the MAPE control loop. Among them, the *hierarchical control pattern* structures the adaptation logic as a hierarchy of MAPE control loops, promising to exploit the benefits of both centralized and decentralized architectures. In [3], we designed a hierarchical approach where the centralized controller issues reconfiguration requests to decentralized managers. In this paper, we consider a different approach, where the centralized controller only provides a feedback to the decentralized managers, which autonomously perform the adaptation actions.

Figure 1 illustrates the deployment controller architecture, highlighting the two-layered approach. The Application Manager and Microservice Managers can work at different time scales. Importantly, the Application Manager provides feedback to each Microservice Manager, which is then taken into account by the Analyze and Plan components of its MAPE loop cycle. At lower-level, we define multiple, decentralized, and autonomous Microservice Managers, each controlling a single microservice using what we call a *local policy*. The *Monitor* collects data about the monitored microservice (i.e., response time and resource utilization). Then, the local *Analyzer* processes the monitored data and determines whether an adaptation action is needed. If an updated is required, the *Planner* identifies which adaptation action is beneficial according to the local policy. Finally, the *Executor* enacts the deployment changes. Exploiting a broader system view, the high-level Application Manager steers the overall adaptation by providing guidelines to the lower levels through a *global policy*. First, it monitors the application performance (i.e., response time) and retrieves the microservice QoS metrics. After their analysis, it uses its global policy to estimate the relative contribution of each microservice to the overall application performance. This information is then forwarded to the Microservice Managers, which can accordingly update the microservices deployment in parallel.

IV. Local Threshold-based Scaling Policy

At the local control level, our goal is to rely on dynamic thresholds and establish a method for automatically adapting their value at run-time, so to efficiently scale each microservice. We use reinforcement learning to learn the scaling threshold adaptation strategy. A RL agent learns what to do (i.e., how to map situations to actions) through direct interaction with the system [22]. It aims to learn an optimal adaptation strategy, so to minimize a numerical cost signal. To minimize the obtained cost, a 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 in RL is to find at run-time a good trade-off between the exploration and exploitation phases.

The Microservice Manager local policy implements the Analyze and Plan steps of the decentralized MAPE loops. For each Microservice Manager, we consider a RL agent in charge of adapting at run-time the scale-out threshold for the controlled microservice, aiming to minimize a long-term cost. In this work, we do not dynamically update scale-in thresholds; nevertheless, the proposed methodology can be easily extended to account also for these thresholds. The RL agent interacts with the microservice in discrete time steps. At each time step, the agent observes the microservice state and performs an action. One time step later, the microservice 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, hence are stochastic. 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. To update the scale-out threshold, given the system state s, the agent performs the action a that minimizes $Q(s, a)$. By observing the incurred immediate costs, $Q(s, a)$ is updated over time, thus improving the threshold update policy.

State. For each microservice m, we define its state at time i as $s_i = (\theta_i, u_i)$, where θ_i is the scale-out threshold, and u_i is the average CPU utilization of the microservice. We denote by $\mathcal{S}$ the set of all the microservice states. Being CPU utilization (u_i) a real number, we discretize it by assuming that $u_i \in \{0, \bar{u}, ..., L\bar{u}\}$, where $\bar{u}$ is a suitable quantum and $L \in \mathbb{N}$ s.t. $L\bar{u} = 1$. We also assume that the scale-out threshold θ_i ranges in the interval $[\Theta_{\text{min}}, \Theta_{\text{max}}]$, where $0 < \Theta_{\text{min}} \leq \Theta_{\text{max}} < 1$.

Action. According to an action selection policy (e.g., ϵ-greedy), the RL agent identifies the threshold adaptation action to be perform. 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. Formally, the action model consists of $\mathcal{A} = \{-\delta, 0, \delta\}$, where $\delta \in (0, 1)$ is a suitable threshold quantum. In particular, $\pm\delta$ represents a threshold adaptation action (i.e., $+\delta$ to add a threshold quantum and $-\delta$ to remove a threshold quantum), and $a = 0$ is the *do nothing* decision. Obviously, not all the actions are available in any microservice state: an action a is valid in a state $s = (\theta, u)$ if $\Theta_{\text{min}} \leq \theta + a \leq \Theta_{\text{max}}$.

Cost Function. We define an immediate cost function $c(s, a, s')$ to capture the cost of carrying out action a when the microservice state transits from s to s'. The RL agent wants to minimize the cost so to jointly satisfy application performance and limit resource wastage. For this purpose, the cost function includes two different contributions:

- the performance penalty c_{perf}, paid whenever the microservice response time t_m is approaching (or exceeds) the response time bound $T_{m,\max}$. This latter parameter is provided by the Application Manager as a function of the overall application response time $T_{\max}$.
- the resource cost c_{res} for running the microservice. 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.

We combine the two cost contributions into a single weighted cost function, where the distinct 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 $[0, 1]$:

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

where:

$$\begin{aligned} c_{\text{perf}}(s, a, s') &= \begin{cases} e^{\xi \frac{t'_m - T_{m,\max}}{T_{m,\max}}} & t'_m \leq T_{m,\max} \\ 1 & \text{otherwise} \end{cases} \\ c_{\text{res}}(s, a, s') &= (1 - \theta') \end{aligned}$$

with ξ is a parameter determining the exponential function steepness, and t'_m is the microservice response time in s'.

Intuitively, the cost function allows us to instruct the Microservice Manager to discriminate between the good system configurations and actions and the bad configurations and actions. As the Microservice Manager aims to minimize the incurred cost, it is encouraged to (i) keep the response time within the given bound and (ii) limit the resource usage. The different weights allow us to express the relative importance of each cost term. We note that this policy only indirectly optimizes the microservice performance: it updates the scaling threshold, which is then used by the Microservice Manager to scale the microservice based on its average CPU utilization.

We also observe that the response time bounds $T_{m,\max}$ grant a share of the global application response time bound $T_{\max}$ to each microservice, accordingly to its relative contribution. The $T_{m,\max}$ terms could be set either statically after preliminary profiling, or dynamically estimated and adapted at run-time by the Application Manager. In Section V, we describe a simple criterium to set these bounds for our reference application.

Q-function Update. To update the Q-function, we consider two RL approaches that differ for the actual learning algorithm adopted and on the assumptions about the system. We first consider the simple model-free Q-learning algorithm that requires no knowledge of the system dynamics. Then, we propose a *model-based* approach, which exploits what is known (or can be estimated) about system dynamics to accordingly update the Q-function and speed-up the learning phase.

1) Q-learning Threshold (QL Threshold): Q-learning is a model-free RL algorithm that does not require a knowledge of the system dynamics. At time i, the Q-learning agent observes the microservice m state s_i and selects a_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., $a_i = \arg\min_{a \in A(s_i)} Q(s_i, a)$) with probability $1 - \epsilon$, whereas 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 using a simple weighted average:

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

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

2) Model-Based Threshold (MB Threshold): The RL agent identifies the threshold adaptation action a_i to perform for microservice m in state s_i relying on a possibly approximated system model. Differently from model-free solutions, the model-based RL approach does not use an action selection policy, but it always selects the best action in term of Q-values, i.e., $a_i = \arg\min_{a \in A(s_i)} Q(s_i, a)$. Moreover, in the model-based RL approach we directly use the Bellman equation to update the Q-function:

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

where we use estimates for the unknown or partially unknown transition probabilities $p(s'|s, a)$ and/or the cost function $c(s, a, s')$, $\forall s, s' \in \mathcal{S}$.[3]

For the estimates of $p(s'|s, a)$, it is sufficient to compute the CPU utilization transition probabilities $P[u_{i+1} = u' | u_i = u]$. Formally:

$$\begin{aligned} p(s'|s, a) &= P[s_{i+1} = (\theta', u') | s_i = (\theta, u), a_i = \delta] \\ &= \begin{cases} P[u_{i+1} = u' | u_i = u] & \theta' = \theta + \delta \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (4)$$

[3]Intuitively, the model-based approach boils down to replacing the model-free equation (2) with one step of the value iteration algorithm using estimates for the unknown parameters.

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. We estimate $p(s'|s, a)$ as the relative number of times the CPU utilization changes from state $j\bar{u}$ to $j'\bar{u}$ in the time interval $\{1, \ldots, i\}$.

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'. As in (1), $c_u(s')$ accounts for the performance penalty. 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 follows:

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

where $\beta \in [0, 1]$ is the *smoothing factor*. $\hat{c}_{u,i}(s')$ is used to compute the cost of applying a in s according to (5). Given a state $s = (\theta, u)$, we observe that in the next state $s' = (\theta', u')$ the expected cost due to $T_{m,\max}$ violation is not lower when the scale-out threshold and/or the CPU utilization increases. Vice versa is also true. Therefore, to speed-up the learning phase, we can heuristically enforce the following properties while updating $\hat{c}_{u,i}(s)$, $\forall s \in \mathcal{S}$:

$$\begin{aligned} \hat{c}_{u,i}(s) &\leq \hat{c}_{u,i}(s') \qquad && \forall \theta \leq \theta', u \leq u' \\ \hat{c}_{u,i}(s) &\geq \hat{c}_{u,i}(s') \qquad && \forall \theta \geq \theta', u \geq u' \end{aligned}$$

V. Global Scaling Policy

Hierarchical policies can scale well in the face of applications composed by a high number of microservices, because of the clear separation of concerns and distribution. Exploiting a system-wide view of the application execution, the Application Manager can easily provide a feedback or proactively notify the different Microservice Managers to improve their cooperation and meet the application requirements.

The Application Manager global policy implements the Analyze and Plan steps of the centralized MAPE loop. Its main goal is to conveniently pinpoint the bottleneck microservices so to not exceed the target application response time $T_{\max}$. To avoid over-complicating the hierarchical policy design, we resort to a simple global policy that dynamically estimates and adapts at run-time the relative microservice contribution to the overall application response time. At each iteration of the MAPE loop and for each microservice m, the Application Manager estimates its target response time $T_{m,\max}$ as $T_{m,\max} = \nu_m \cdot T_{\max}$, where ν_m is the (average) contribution of microservice m to the overall application response time. To determine ν_m for each microservice, the Application Manager uses the application DAG. Hence, ν_m is updated using a simple exponential weighted average:

$$\nu_m \leftarrow (1 - \phi)\nu_m + \phi \frac{\bar{t}_m}{\bar{t}_{\Pi_m}} \tag{7}$$

where $\phi \in [0, 1]$ is the *smoothing factor*, $\bar{t}_m$ is the average microservice response time and $\bar{t}_{\Pi_m} = \frac{\sum_{\pi \in \Pi_m} p_\pi \cdot t_\pi}{\sum_{\pi \in \Pi_m} p_\pi}$, being $\bar{t}_{\Pi_m}$ the weighted average of the response times of all sink-source paths including m with $\Pi_m \subseteq \Pi$, and $p_\pi \in [0, 1]$ the probability that a service request invokes path π. We estimate p_π as the relative number of times the incoming request invokes the microservices belonging to path π. The Application Manager then sends the $T_{m,\max}$ value to the Microservice Manager in charge of controlling m. Relying on its local policy, the Microservice Manager can accordingly update its scaling strategy.

VI. Experimental Results

We evaluate the proposed deployment adaptation solutions by means of simulations. To capture the variability of microservice-based applications, we consider three different types of application DAG, namely sequential, diamond, and complex, as shown in Figure 2. They have the same number of microservices. Within each microservice m, Figure 2 reports its service rate μ_m; on top of each microservice, we show the ratio between the overall outgoing request rate and the incoming request rate; and on the outgoing edges of m, we show the invocation probability of the successor microservices of m, considering a probabilistic microservice invocation [23]. Moreover, we compare the proposed dynamic-threshold approaches (i.e., MB Threshold and QL Threshold) against a static threshold-based policy (i.e., Static Threshold), which represents the most widely adopted auto-scaling solution in container orchestration frameworks (e.g., Kubernetes).

Without lack of generality, at each discrete time step i, we model each microservice as an M/M/k_i queue, where k_i is the number of microservice replicas.

We set the basic service rate μ shown in Figure 2 to 140 requests/s. Each application requires its overall response time to be below $T_{\max} = |\tilde{\Pi}|/\bar{\mu}$ ms, where $|\tilde{\Pi}|$ is the length of the longest application path and $\bar{\mu}$ is the lowest components' service rate: thus, the complex application requires $T_{\max} = 59.5$ ms, the sequential one $T_{\max} = 83.3$ ms, and the diamond one $T_{\max} = 35.7$ ms. We consider that the application receives an incoming request rate that changes over time according to the workload pattern shown in Figure 3. The RL algorithms use the following parameters: $\Theta_{\min} = 0.5$, $\Theta_{\max} = 0.9$, $\xi = 10$, discount factor $\gamma = 0.99$; QL Threshold also uses $\alpha = 0.1$ and $\epsilon = 0.1$ and MB Threshold uses $\beta = 0.1$. For the global policy, we set $\phi = 0.1$. We use small values of the smoothing factors (i.e., α, β, and ϕ) so to weigh more recent samples and improve the agents ability to react to system changes.

To discretize the application state, we use $\bar{u} = 0.1$. The scale-in threshold is set to 20% of CPU utilization. Our simulator is written in Java and uses one class for each microservice that, in turn, is modeled as an M/M/k queue. Two main classes, Application Manager and Microservice Manager, implement the layered MAPE loop. At each time step, the simulator calls the MAPE components of the Application Manager and, then, of the Microservice Managers. The Application Manager

TABLE I: Application performance using different threshold-based scaling policies.

Topology	*Policy*	*Configuration*	*Average threshold value (%)*	*Standard deviation of the threshold value*	*Threshold adaptations (%)*	*Median response time (ms)*	*$T_{\max}$ violations (%)*	*Average CPU utilization (%)*	*Average replicas per service*
complex	**MB Threshold**	$w_{\text{perf}} = 1$, $w_{\text{res}} = 0$	50.21	1.30	2.29	33.17	0.05	30.81	2.44
		$w_{\text{perf}} = 0.50$, $w_{\text{res}} = 0.50$	89.44	2.64	4.39	38.90	2.62	40.70	1.82
		$w_{\text{perf}} = 0$, $w_{\text{res}} = 1$	89.96	0.57	1.02	41.95	17.62	44.16	1.70
	QL Threshold	$w_{\text{perf}} = 1$, $w_{\text{res}} = 0$	71.35	12.36	61.18	35.76	1.57	34.35	2.21
		$w_{\text{perf}} = 0.50$, $w_{\text{res}} = 0.50$	76.72	11.51	62.22	35.42	1.34	35	2.15
		$w_{\text{perf}} = 0$, $w_{\text{res}} = 1$	76.99	11.47	62.37	34.87	2.70	35.15	2.15
	Static Threshold	50% of CPU utilization	50	0	0	33.14	0.05	30.78	2.44
		60% of CPU utilization	60	0	0	35.31	0.70	34.82	2.15
		70% of CPU utilization	70	0	0	37.10	2.15	38.30	1.95
		80% of CPU utilization	80	0	0	27.02	19.92	37.02	1.52
diamond	**MB Threshold**	$w_{\text{perf}} = 1$, $w_{\text{res}} = 0$	50.17	1.17	0.99	22.75	0.12	28.17	2.02
		$w_{\text{perf}} = 0.50$, $w_{\text{res}} = 0.50$	87.91	6.69	5.34	25.35	3.05	34.04	1.64
		$w_{\text{perf}} = 0$, $w_{\text{res}} = 1$	89.96	0.57	0.94	33.22	45.54	39.86	1.41
	QL Threshold	$w_{\text{perf}} = 1$, $w_{\text{res}} = 0$	69.73	12.10	63.76	23.62	1.62	29.84	1.91
		$w_{\text{perf}} = 0.50$, $w_{\text{res}} = 0.50$	76.12	11.68	62.89	23.82	2.10	30.60	1.87
		$w_{\text{perf}} = 0$, $w_{\text{res}} = 1$	76.92	11.49	62.67	24.12	3.07	30.90	1.84
	Static Threshold	50% of CPU utilization	50	0	0	22.75	0.12	28.16	2.02
		60% of CPU utilization	60	0	0	24.40	1.12	31.80	1.79
		70% of CPU utilization	70	0	0	26.29	7.62	35.27	1.61
		80% of CPU utilization	80	0	0	27.02	19.92	37.02	1.52
sequence	**MB Threshold**	$w_{\text{perf}} = 1$, $w_{\text{res}} = 0$	50.11	1.17	1.34	46.80	0.05	32.79	3.92
		$w_{\text{perf}} = 0.50$, $w_{\text{res}} = 0.50$	86.35	5.76	9.07	53.48	0.65	43.15	2.94
		$w_{\text{perf}} = 0$, $w_{\text{res}} = 1$	89.96	0.56	0.92	60.98	16.82	49.64	2.60
	QL Threshold	$w_{\text{perf}} = 1$, $w_{\text{res}} = 0$	69.73	12.10	63.76	47.91	1.62	29.84	1.91
		$w_{\text{perf}} = 0.50$, $w_{\text{res}} = 0.50$	75.98	11.67	63.16	48.40	0.45	37.34	3.52
		$w_{\text{perf}} = 0$, $w_{\text{res}} = 1$	77.40	11.29	62.32	48.69	2.15	38.11	3.47
	Static Threshold	50% of CPU utilization	50	0	0	33.14	0.05	30.78	2.44
		60% of CPU utilization	60	0	0	35.31	0.70	34.82	2.15
		70% of CPU utilization	70	0	0	37.10	2.15	38.30	1.95
		80% of CPU utilization	80	0	0	54.64	3.45	44.81	2.85

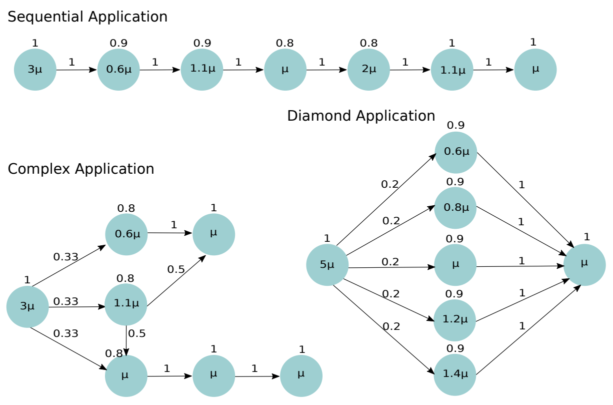


Fig. 2: Sequential, diamond and complex applications.

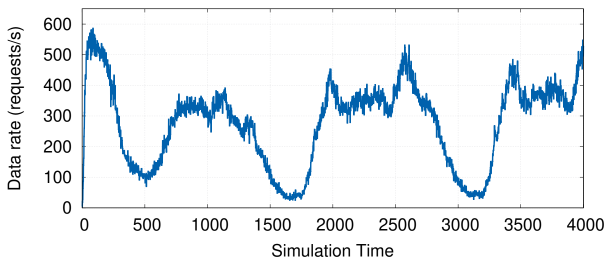
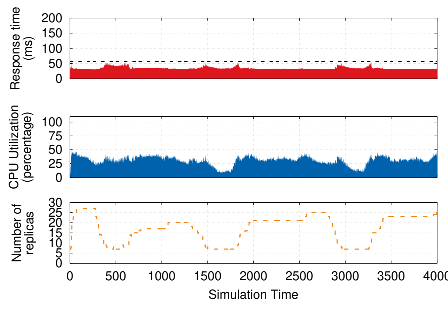


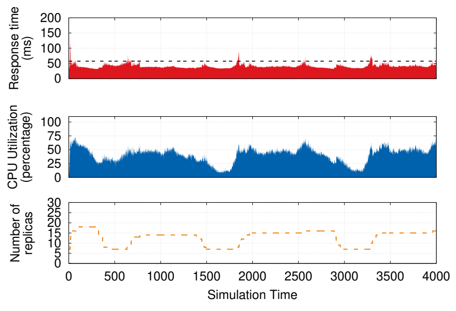
Fig. 3: Workload used for the reference applications.

communicates with the Microservice Managers only to update the $T_{m,\max}$ terms, at most once per time step. We execute all simulations on a machine equipped with Intel Core i7-8550U and 16 GB of RAM. Table I summarizes the simulation results.

A. Different Application Topologies and Configurations

The first set of experiments aims to show the flexibility of RL-based solutions to dynamically adapt the scaling thresholds for microservice-based applications. Due to space limitations, we mainly focus the discussion on the complex application; nevertheless, Table I reports the results for the other topologies, as well. Overall, we can see that, for a specific topology, the application performances change under the different scale-out threshold policies. The Static Threshold policy is application-unaware and not flexible, meaning that it is not easy to satisfy QoS requirements of latency-sensitive applications by setting a threshold on CPU utilization. From Table I, we can observe that small threshold changes may lead to a significant performance deterioration. Conversely, the dynamic thresholds can be trained to optimize different deployment objectives, e.g., see the response time median, $T_{\max}$ violations, or the average number of replicas. The model-based RL solution can successfully learn a different strategy to update the scaling thresholds for the different application topologies. We note that, although the thresholds have similar value for all the topologies, their standard deviation changes, especially for multi-objective optimizations. This indicates a diversification of the thresholds per application components, whose value has been empirically determined by the RL agents. Figure 4 shows the application performance during the whole experiment, when the model-based solution updates the scaling thresholds for each application microservice (i.e., MB Threshold is used). We can see that the application has a different performance when different weights for the cost function are used (Eq. 1). When the cost function penalizes response time violations (i.e., with $w_{\text{perf}} = 1$), the median of the application response time is 33 ms (0% of $T_{\max}$ violations) and, on average, each microservice runs with 2.4 replicas. Conversely, when we aim to save resources (i.e., $w_{\text{res}} = 1$), the application response time median grows to 42 ms (18% of $T_{\max}$ violations) and, on average, each microservice runs with 1.7 replicas. The different behavior is also clear by comparing the overall number of microservices replicas during the


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


(b) Weights: $w_{\text{res}} = 0.5$ and $w_{\text{perf}} = 0.5$.

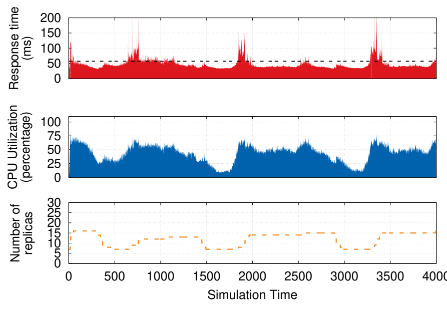

(c) Weights: $w_{\text{res}} = 1$ and $w_{\text{perf}} = 0$.

Fig. 4: Performance for complex application using MB Threshold.

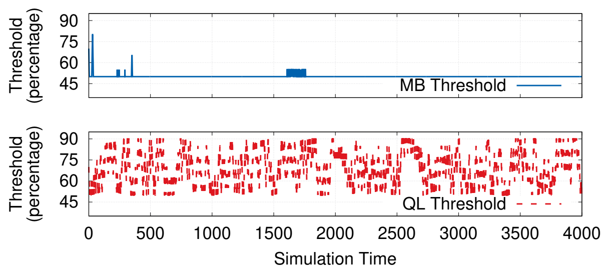


Fig. 5: Scale-out threshold updates computed by the RL-based heuristics for the first microservice of the complex application.

experiment (see Figures 4a and 4c). Figure 4c also shows that the application response time follows the incoming workload, with different response time peaks when CPU utilization is approaching 75%. On the other hand, Figure 4a shows the benefits of replication: since the application is readily scaled, the resulting response time is below $T_{\max}$ and has also a reduced variance.

Besides the weight configurations at the opposite ends, we can obtain a wide set of adaptation strategies that differ by the relative importance of the two deployment goals. Here, we propose a simple case, where we set $w_{\text{perf}} = 0.50$ and $w_{\text{res}} = 0.50$. The median application response time is 39 ms, with about 3% of $T_{\max}$ violations. In this case, we obtain an average threshold value that is rather close to the case of $w_{\text{perf}} = 1$, even though there is a higher variance due to the different thresholds computed for the different application microservices.

The MB Threshold policy is flexible enough to host different sets of weights, thus allowing to explore diverse trade-offs between improving performance and saving resource. Importantly, the RL approach can automatically learn the most suitable strategy to satisfy the user preferences, estimating the mapping between user- and system-oriented metrics.

B. *Comparing QL and MB Thresholds*

We now compare the model-based RL approach against the simple and model-free Q-learning solution. Both the RL strategies are used to update the scale-out thresholds in a distributed manner, for each application microservice. The two approaches resort on the cost function (Eq. 1) to receive a feedback of the performed action and update their Q-value estimations. To visualize the update of the scaling threshold by the two RL policies, we report in Figure 5 the threshold value for the first microservice (with 3μ) of the complex application, when we want to optimize the performance ($w_{\text{perf}} = 1$). Intuitively, the best threshold should be the lowest possible, so to use as many replicas as possible. The model-based RL solution benefits from the system model to quickly learn how to update the scaling thresholds. This holds true for all the application topologies and cost function configurations (see Table I). Conversely, Q-learning continuously updates the scaling thresholds, meaning that it is still exploring the best actions to perform. This behavior is also reflected on the application response time, whose median value does not change under the different cost function configurations.

C. *Hierarchical Application Control*

In this section, we investigate the global policy, used to provide an adaptation feedback to the decentralized RL agents.

When a static threshold policy is used to scale a complex application, we usually set a single threshold value for all the application components. Tuning different thresholds for the different components is costly, so it is not usually done in practice. Conversely, the cooperation between the Application Manager and the decentralized Microservice Managers allows to automatically adapt the thresholds for the different application components, according to their run-time behavior. Figure 6 reports the utilization, scale-out threshold, and number of replicas of two components of the diamond application, microservice 1 (with 0.6μ) and microservice 5 (with 1.4μ). In this case, the MB Threshold policy computes the thresholds under the weights configuration $w_{\text{perf}} = w_{\text{res}} = 0.5$. We observe that the two microservices have a different degree of replication, with the bottleneck component, microservice 1, running with more replicas than microservice 5 (on average, 2 and 1, respectively). Apparently, the RL agent prefers to use a high value as the threshold rest value, which is promptly updated when scaling actions are needed or the application runs under heavy workload conditions (see the time interval between 2000 and 3000 time units). To further investigate the

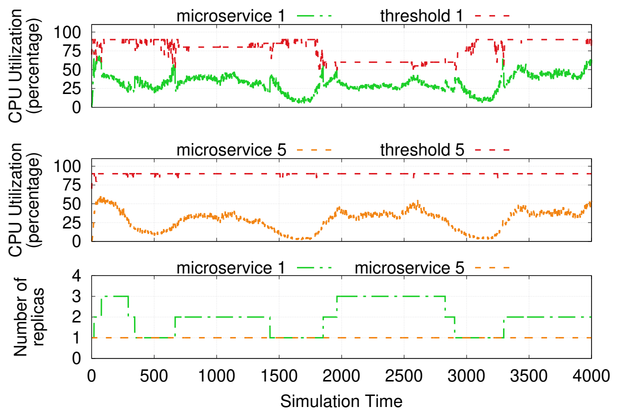


Fig. 6: Threshold diversification for microservices with service rate 0.6μ and 1.4μ of the diamond application. Thresholds computed using model-based RL and $w_{\text{perf}} = w_{\text{res}} = 0.5$.

global policy, we run another experiment where we turned off the Application Manager and statically set $T_{m,\max} = T_{\max}/3$, $\forall m \in M$. As a result, the median application response time is 31 ms, violating $T_{\max}$ 37% of the time (instead of 25 ms and 3% of bound violations as in the previous setting). We observe that the distributed RL agents more slowly learn to distinguish the bottleneck components and diversify the thresholds: e.g., the bottleneck component runs with 1.6 replicas (instead of 2). The Application Manager helps to capture the heterogeneity of the application microservices, leading to the definition of different thresholds that better optimize performance and avoid resource over-/under-provisioning.

VII. Conclusions

In this paper, we presented a novel self-adaptive threshold-based policy for scaling microservice-based applications. Specifically, we designed a two-layered hierarchical control architecture where, at the lower level, decentralized controllers scale microservices using dynamic thresholds and, at the higher level, a centralized controller analyzes the relative microservice contribution to the overall application performance. To update the scaling thresholds at run-time, we rely on model-free and model-based RL algorithms, obtaining QL Threshold and MB Threshold, respectively. As regards the global policy, we proposed a simple yet effective heuristic to empirically estimate the microservice contribution to the application performance. Using simulation, we evaluated the proposed solutions and compared them against a static threshold-based policy, which is the most widely adopted scaling strategy. While the QL Threshold policy suffers from slow learning rate, our MB Threshold clearly outperforms all the other approaches. Differently from a static threshold-based approach, the MB solution not only differentiates the scaling threshold for the different microservices, but can also improves the policy flexibility, because it can learn different threshold update strategies according to the deployment goals.

As future work, we will integrate the proposed solutions in Kubernetes, so to evaluate them in a real environment. Moreover, we plan to design novel hierarchical policies that can jointly control the scaling and placement of microservice-based applications in a geo-distributed computing environment.

References

[1] 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.

[2] N. Cruz Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet of Things J.*, vol. 7, no. 5, pp. 4195–4202, 2020.

[3] F. Rossi, V. Cardellini, and F. Lo Presti, "Hierarchical scaling of microservices in Kubernetes," in *Proc. of IEEE ACSOS '20*, 2020, pp. 28–37.

[4] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *J. Netw. Comput. Appl.*, vol. 160, 2020.

[5] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Trans. Cloud Comput.*, 2020.

[6] A. Beloglazov and R. Buyya, "Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers," in *Proc. of MGC '10*. ACM, 2010.

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

[8] 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.

[9] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 155–162, 2012.

[10] 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.

[11] C. Kan, "DoCloud: An elastic cloud platform for web applications based on Docker," in *Proc. of ICACT '16*. IEEE, 2016, pp. 478–483.

[12] D. Breitgand, M. Goldstein, E. Henis, and O. Shehory, "Efficient control of false negative and false positive errors with separate adaptive thresholds," *IEEE Trans. Netw. Service Manag.*, vol. 8, no. 2, 2011.

[13] V. Persico, D. Grimaldi, A. Pescapè, A. Salvi, and S. Santini, "A fuzzy approach based on heterogeneous metrics for scaling out public clouds," *IEEE Trans. Parallel Distrb. Syst.*, vol. 28, no. 8, pp. 2117–2130, 2017.

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

[15] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *Proc. of IEEE ICDCS '19*, 2019, pp. 1994–2004.

[16] 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.

[17] S. M. R. Nouri, H. Li, S. Venugopal, W. Guo *et al.*, "Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications," *Future Gener. Comput. Syst.*, vol. 94, 2019.

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

[19] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn *et al.*, "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. of EuroSys '19*. ACM, 2019.

[20] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies," *ACM Queue*, vol. 16, no. 4, 2018.

[21] 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.

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

[23] D. A. Menasce, "Composing web services: A QoS view," *IEEE Internet Comput.*, vol. 8, no. 6, pp. 88–90, 2004.