Auto-scaling Policies to Adapt the Application Deployment in Kubernetes

Fabiana Rossi

Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy
f.rossi@ing.uniroma2.it

Abstract The ever increasing diffusion of computing devices enables a new generation of containerized applications that operate in a distributed cloud environment. Moreover, the dynamism of working conditions calls for an elastic application deployment, which can adapt to changing workloads. Despite this, most of the existing orchestration tools, such as Kubernetes, include best-effort threshold-based scaling policies whose tuning could be cumbersome and application dependent. In this paper, we compare the default threshold-based scaling policy of Kubernetes against our model-based reinforcement learning policy. Our solution learns a suitable scaling policy from the experience so to meet Quality of Service requirements expressed in terms of average response time. Using prototype-based experiments, we show the benefits and flexibility of our reinforcement learning policy with respect to the default Kubernetes scaling solution.

Keywords: Kubernetes · Elasticity · Reinforcement Learning · Self-adaptive systems.

1 Introduction

Elasticity allows to adapt the application deployment at run-time in face of changing working conditions (e.g., incoming workload) and to meet stringent Quality of Service (QoS) requirements. Exploiting operating system level virtualization, *software containers* allow to simplify the deployment and management of applications, also offering a reduced computational overhead with respect to virtual machines. The most popular container management system is Docker. It allows to simplify the creation, distribution, and execution of applications inside containers. Although the container management systems can be used to deploy simple containers, managing a complex application (or multiple applications) at run-time requires an *orchestration tool*. The latter automates container provisioning, management, communication, and fault-tolerance. Although several orchestration tools exist [5,8], *Kubernetes*[1], an open-source platform introduced by Google in 2014, is the most popular solution. Kubernetes includes a Horizontal

[1] https://kubernetes.io

Pod Autoscaler enabling to automatically scale the application deployment using a threshold-based policy based on cluster-level metrics (i.e., CPU utilization). However, this threshold-based scaling policy is not well suited to satisfy QoS requirements of latency-sensitive applications. Determining a suitable threshold is cumbersome, requiring to identify the relation between a system metric (i.e., utilization) and an application metric (i.e., response time), as well as to know the application bottleneck (e.g., in terms of CPU or memory). In this paper, we compare the default threshold-based scaling policy of Kubernetes against model-free and model-based reinforcement learning policies [14]. Our model-based solution automatically learns a suitable scaling policy from the experience so to meet QoS requirements expressed in term of average response time. To perform such comparison, we use our extension of Kubernetes, which includes a more flexible autoscaler that can be easily equipped with new scaling policies. The remainder of the paper is organized as follows. In Section 2, we discuss related works. In Section 3, we describe the Kubernetes features. Then, we propose a reinforcement learning-based scaling policy to adapt at run-time the deployment of containerized applications (Section 4). In Section 5, we evaluate the proposed solutions using prototype-based experiments. We show the flexibility and efficacy of using a reinforcement learning solution compared to the default Kubernetes scaling policy. In Section 6, we outline the ongoing and future research directions.

2 Related Work

The elasticity of containers is carried out in order to achieve different objectives: to improve application performance (e.g., [4]), load balancing and resource utilization (e.g., [1,11]), energy efficiency (e.g., [3]), and to reduce the deployment cost (e.g., [6,2]). Few works also consider a combination of deployment goals (e.g., [18]). Threshold-based policies are the most popular approaches to scale containers at run-time (e.g., [4,10]). Also the noteworthy orchestration tools (e.g., Kubernetes, Docker Swarm, Amazon ECS, and Apache Hadoop YARN) usually rely on best-effort threshold-based scaling policies based on some cluster-level metrics (e.g., CPU utilization). However, all these approaches require a non-trivial manual tuning of the thresholds, which can also be application-dependent. To overcome to this issue, solutions in literature propose container deployment methods ranging from mathematical programming to machine learning solutions. The mathematical programming approaches exploit methods from operational research in order to solve the application deployment problem (e.g., [12,13,18]). Since such a problem is NP-hard, other efficient solutions are needed. In the last few years, reinforcement learning (RL) has become a widespread approach to solve the application deployment problem at run-time. RL is a machine learning technique by which an agent can learn how to make (scaling) decisions through a sequence of interactions with the environment [15]. Most of the existing solutions consider the classic model-free RL algorithms (e.g., [7,16,17]), which however suffer from slow convergence rate. To tackle this issue, in [14], we propose a novel model-based RL solution that exploits what is known (or can be estimated) about the system dynamics to adapt the application deployment at run-time. Experimental results based on Docker Swarm have shown the flexibility of our approach, which can learn different adaptation strategies according to the optimized deployment objectives (e.g., meet QoS requirements in terms of average response time). Moreover, we have shown that the model-based RL agent learns a better adaptation policy than other model-free RL solutions. Encouraged by the previous promising results, in this paper, we integrate the model-based RL solution in Kubernetes, one of the most popular container orchestration tools used in the academic and industrial world. Experimental results in [8] demonstrate that Kubernetes performs better than other existing orchestration tools, such as Docker Swarm, Apache Mesos, and Cattle. However, Kubernetes is not suitable for managing latency-sensitive applications in a extremely dynamic environment. It is equipped with a static best-effort deployment policy that relies on system-oriented metrics to scale applications in face of workload variations. In this paper, we first extend Kubernetes to easily introduce self-adaptation capabilities. Then, we integrate RL policies in Kubernetes and compare them against the default Kubernetes auto-scaling solution.

3 Kubernetes

Kubernetes is an open-source orchestration platform that simplifies the deployment, management, and execution of containerized applications. Based on a master-worker decentralization pattern, it can replicate containers for improving resource usage, load distribution, and fault-tolerance. The master node maintains the desired state at run-time by orchestrating applications (using pods). A worker is a computing node that offers its computational capability to enable the execution of pods in distributed manner. A pod is the smallest deployment unit in Kubernetes. When multiple containers run within a pod, they are co-located and scaled as an atomic entity. To simplify the deployment of applications, Kubernetes introduces Deployment Controllers that can dynamically create and destroy pods, so to ensure that the desired state (described in the *deployment* file) is preserved at run-time. Kubernetes also includes a Horizontal Pod Autoscaler[2] to automatically scale the number of pods in a Deployment based on the ratio between the target value and the observed value of pod's CPU utilization. Setting the CPU utilization threshold is a cumbersome and error-prone task and may require a knowledge of the application resource usage to be effective.

To address this limitation, we equip Kubernetes with a decentralized control loop. In a single loop iteration, it monitors the environment and the containerized applications, analyzes application-level (i.e., response time) and cluster-level (i.e., CPU utilization) metrics, and plans and executes the corresponding scaling actions. The modularity of the control loop allows us to easily equip it with different QoS-aware scaling policies. To dynamically adapt the application deployment according to the workload variations, we consider RL policies.

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

4 Reinforcement Learning Scaling Policy

Differently from the Kubernetes scaling policy, we aim to design a flexible solution that can be easily customized by manually tuning various configuration parameters. In this paper, we customize the RL solution proposed in [14] to scale at run-time the number of application instances (i.e., pods). RL refers to a collection of trial-and-error methods by which an 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*). In a single control loop iteration, the RL agent selects the adaptation action to be performed. As first step, according to the received application and cluster-oriented metrics, the RL agent determines the Deployment Controller state and updates the expected long-term cost (i.e., Q-function). We define the application state as $s = (k, u)$, where k is the number of application instances (i.e., pods), and u is the monitored CPU utilization. We denote by $\mathcal{S}$ the set of all the application states. We assume that $k \in \{1, 2, ..., K_{\max}\}$; being the CPU utilization (u) a real number, we discretize it by defining that $u \in \{0, \bar{u}, ..., L\bar{u}\}$, where $\bar{u}$ is a suitable quanta. For each state $s \in \mathcal{S}$, we define the set of possible adaptation actions as $\mathcal{A}(s) \subseteq \{-1, 0, 1\}$, where ± 1 defines a scaling action (i.e., $+1$ to scale-out and -1 to scale-in), and 0 is the *do nothing* decision. Obviously, not all the actions are available in any application state, due to the upper and lower bounds on the number of pods per application (i.e., $K_{\max}$ and 1, respectively). Then, according to an action selection policy, the RL agent identifies the scaling action a to be performed in state s. The execution of a in s leads to the transition in a new application state (i.e., s') and to the payment of an immediate cost. We define the immediate cost $c(s, a, s')$ as the weighted sum of different terms, such as the *performance penalty*, c_{perf}, *resource cost*, c_{res}, and *adaptation cost*, c_{adp}. We normalized them in the interval $[0, 1]$, where 0 represents the best value (no cost), 1 the worst value (highest cost). Formally, we have: $c(s, a, s') = w_{\text{perf}} \cdot c_{\text{perf}} + w_{\text{res}} \cdot c_{\text{res}} + w_{\text{adp}} \cdot c_{\text{adp}}$, where w_{adp}, w_{perf} and w_{res}, $w_{\text{adp}} + w_{\text{perf}} + w_{\text{res}} = 1$, are non negative weights that allow us to express the relative importance of each cost term. We can observe that the formulation of the immediate cost function $c(s, a, s')$ is general enough and can be easily customized with other QoS requirements. The *performance penalty* is paid whenever the average application response time exceeds the target value $R_{\max}$. The *resource cost* is proportional to the number of application instances (i.e., pods). The *adaptation cost* captures the cost introduced by Kubernetes to perform a scaling operation. The traffic routing strategy used in Kubernetes forwards the application requests to the newly added pod, even if not all containers in the pod are already running. We observe that, for this reason, we prefer horizontal scaling to vertical scaling operations. When a vertical scaling changes a pod configuration (e.g., to update its CPU limit), Kubernetes spawns new pods as a replacement of those with the old configuration. In this phase, the application availability decreases and only a subset of the incoming requests are processed. Conversely, a scale-out action introduces a reduced adaptation cost inversely proportional to the number of application instances.

The received immediate cost contributes to update the Q-function. 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. The existing RL policies differ in how they update the Q-function and select the adaptation action to be performed (i.e., action selection policy) [15]. To adapt the application deployment, we consider a model-based solution which we have extensively evaluated in [14]. At any decision step, the proposed model-based RL solution does not use an action selection policy (e.g., ϵ-greedy action selection policy) but it always selects the best action in term of Q-values, i.e., $a = \arg\min_{a' \in A(s)} Q(s, a')$. Moreover, to update the Q-function, the simple weighted average of the traditional RL solutions (e.g., Q-learning) is replaced by the Bellman equation [15]:

$$Q(s,a) = \sum_{s' \in \mathcal{S}} p(s'|s,a) \left[c(s,a,s') + \gamma \min_{a' \in \mathcal{A}} Q(s',a') \right] \quad {\scriptstyle \forall s \in \mathcal{S}, \atop \forall a \in \mathcal{A}(s)} \tag{1}$$

where $\gamma \in [0, 1)$ is the *discount factor*, $p(s'|s, a)$ and $c(s, a, s')$ are, respectively, the transition probabilities and the cost function $\forall s, s' \in \mathcal{S}$ and $a \in A(s)$. Thanks to the experience, the proposed model-based solution is able to maintain an empirical model of the unknown external system dynamics (i.e., $p(s'|s, a)$ and $c(s, a, s')$) speeding-up the learning phase. Further details on our model-based RL solution can be found in [14].

5 Results

We show the self-adaptation capabilities of Kubernetes when equipped with model-free and model-based RL policies as well as the default threshold-based solution (by the Horizontal Pod Autoscaler). The RL solutions scale pods using user-oriented QoS attributes (i.e., response time), whereas the Horizontal Pod Autoscaler uses a best-effort threshold-based policy based on cluster-level metrics (i.e., CPU utilization). The evaluation uses a cluster of 4 virtual machines of the Google Cloud Platform; each virtual machine has 2 vCPUs and 7.5 GB of RAM (type: n1-standard-2). We consider a reference CPU-intensive application that computes the sum of the first n elements of the Fibonacci sequence. As shown in Figure 1, the application receives a varying number of requests. It follows the workload of a real distributed application [9], accordingly amplified

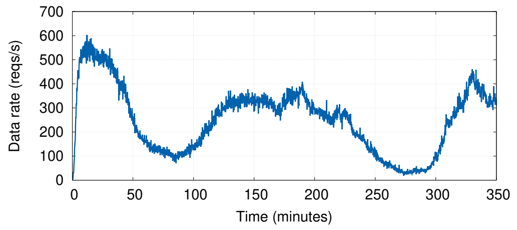
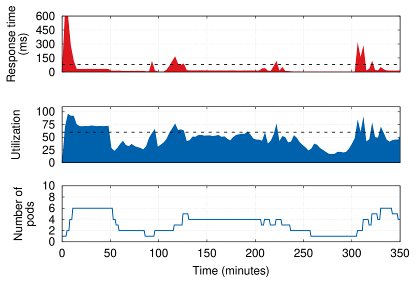
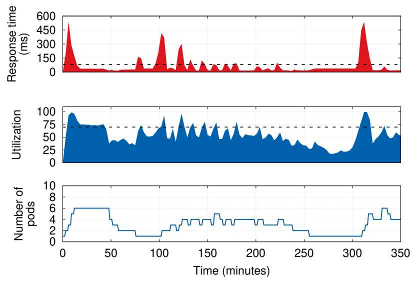


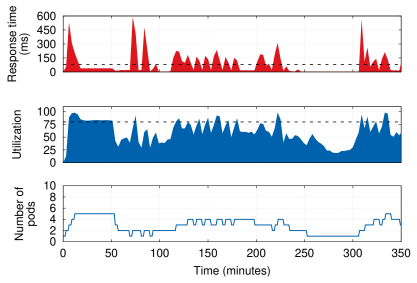
Figure 1: Workload used for the reference application.



(a) Threshold at 60% of CPU utilization.



(b) Threshold at 70% of CPU utilization.

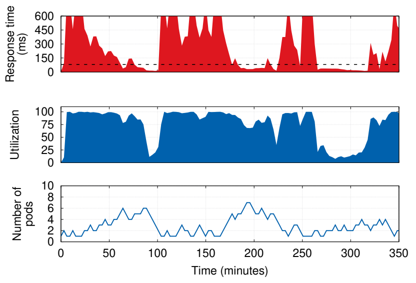


(c) Threshold at 80% of CPU utilization.

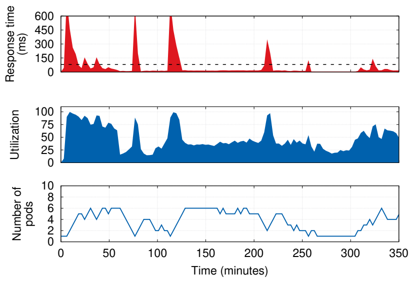
Figure 2: Application performance using Horizontal Pod Autoscaler.

and accelerated so to further stress the application resource requirements. The application expresses the QoS in terms of target response time $R_{\max} = 80$ ms. To meet $R_{\max}$, it is important to accordingly adapt the number of application instances. The Kubernetes autoscaler executes a control loop every 3 minutes. To learn an adaptation policy, we parameterize the model-based RL algorithm as in our previous work [14]. For sake of comparison, we consider also the model-free Q-learning approach that chooses a scaling action according to the ϵ-greedy selection policy: at any decision step, the Q-learning agent chooses, with probability ϵ, a random action, whereas, with probability $1 - \epsilon$, it chooses the best known action. For Q-learning, we set ϵ to 10%. To discretize the application state, we use $K_{\max} = 10$ and $\bar{u} = 0.1$. For the immediate cost function, we consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$. This weight configuration allows to optimize the application response time, considered to be more important than saving resources and reducing the adaptation costs.

The default Kubernetes threshold-based scaling 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 (see Figures 2a–2c). From Table 1, we can observe that small changes in the threshold setting lead to a significant performance deterioration. Setting the scaling threshold is



(a) Q-learning.



(b) Model-based RL.

Figure 3: Application performance using RL policies.

Table 1: Application performance under the different scaling policies.

Elasticity Policy	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average number of pods*	*Median of Response time (ms)*	*Adaptations* (%)
Model-based	14.40	48.51	3.75	16.38	56.00
Q-learning	64.0	76.94	2.90	201.71	65.6
HPA thr = 60	9.20	50.81	3.54	16.11	6.38
HPA thr = 70	21.43	54.43	3.14	34.61	10.96
HPA thr = 80	40.12	63.70	3.18	37.54	12.89

cumbersome, e.g., with threshold on 80% of CPU utilization, we obtain a rather high number of $R_{\max}$ violations. With the scaling threshold at 70% of CPU utilization, the application violates $R_{\max}$ 21% of time, with 54% of average CPU utilization. With the scaling threshold at 60% of CPU utilization, the application has better performance ($R_{\max}$ is exceeded only 9% of time), even though we might still perform a finer threshold tuning to further increase it.

Conversely, the RL approach is general and more flexible, requiring only to specify the desired deployment objectives. It allows to indicate *what* the user aims to obtain (through the cost function weights), instead of *how* it should be obtained. In particular, a RL learning agent learns the scaling policy in an automatic manner. Figures 3a and 3b show the application performance when the model-free and model-based RL solutions are used. The RL agent starts with no knowledge on the adaptation policy, so it begins to explore the cost of each adaptation action. When Q-learning is used, the RL agent slowly learns how to adapt the application deployment. As we can see from Figure 3a and Table 1, the application deployment is continuously updated (i.e., 66% of the time) and the RL agent does not learn a good adaptation policy within the experiment duration. As a consequence, the application response time exceeds $R_{\max}$ most of the time. Taking advantage of the system knowledge, the model-based solution has a very different behavior: it obtains better performance and more quickly reacts to workload variations. We can see that, in the first minutes of the experiment, the model-based solution does not always respect the target application response time. However, as soon as a suitable adaptation policy is learned, the model-based RL solution can successfully scale the application and meet the application response time requirement most of the time. The learned adaptation policy deploys a number of pods that follows the application workload (see Figures 1 and 3b), maintaining a reduced number of $R_{\max}$ violations (14.4%) and a good average resource utilization (49%).

We should observe that, even though a fine grained threshold tuning can be performed (thus improving performance of the default Kubernetes scaling policy), the RL-based approach automatically learns a suitable and satisfying adaptation strategy. Moreover, changing the cost function weights, the RL solution can easily learn different scaling policies, e.g., to improve resource utilization or to reduce deployment adaptations [14].

6 Conclusion

Kubernetes is one of the most popular orchestration tools to manage containers in a distributed environment. To react to workload variations, it includes a threshold-based scaling policy that changes the application deployment according to cluster-level metrics. However, this approach is not well-suited to meet stringent QoS requirements. In this paper, we compare model-free and model-based RL scaling policies against the default threshold-based solution. The prototype-based results have shown the flexibility and benefits of RL solutions: while the model-free Q-learning suffers from slow convergence time, the model-based approach can successfully learn the best adaptation policy, according to the user-defined deployment goals.

As future work, we plan to investigate the deployment of applications in geo-distributed environment, including edge/fog computing resources located at the network edges. The default Kubernetes scheduler spreads containers on computing resources not taking into account the not-negligible network delay among them. This can negatively impact the performance of latency-sensitive applications. Therefore, alongside the elasticity problem, also the placement problem (or scheduling problem) should be efficiently solved at run-time. We want to extend the proposed heuristic so to efficiently control the scaling and placement of multi-component applications (e.g., micro-services). When an application consists of multiple components that cooperate to accomplish a common task, adapting the deployment of a component impacts on performance of the other components. We are interested in considering the application as a whole, so to develop policies that can adapt, in proactive manner, the deployment of inter-connected components, avoiding performance penalties.

Acknowledgment

The author would like to thank her supervisor Prof. Valeria Cardellini and to acknowledge the support by Google with the GCP research credits program.

References

1. Abdelbaky, M., Diaz-Montes, J., Parashar, M., Unuvar, M., Steinder, M.: Docker containers across multiple clouds and data centers. In: Proc. of IEEE/ACM UCC 2015. pp. 368–371 (2015)
2. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Autonomic vertical elasticity of Docker containers with ElasticDocker. In: Proc. of IEEE CLOUD '17. pp. 472–479 (2017)
3. Asnaghi, A., Ferroni, M., Santambrogio, M.D.: DockerCap: A software-level power capping orchestrator for Docker containers. In: Proc. of IEEE EUC '16. pp. 90–97 (2016)
4. Barna, C., Khazaei, H., Fokaefs, M., Litoiu, M.: Delivering elastic containerized cloud applications to enable DevOps. In: Proc. of SEAMS '17. pp. 65–75 (2017)
5. Casalicchio, E.: Container orchestration: A survey. In: Systems Modeling: Methodologies and Tools, pp. 221–235. Springer International Publishing, Cham (2019)
6. Guan, X., Wan, X., Choi, B.Y., Song, S., Zhu, J.: Application oriented dynamic resource allocation for data centers using Docker containers. IEEE Commun. Lett. **21**(3), 504–507 (2017)
7. Horovitz, S., Arian, Y.: Efficient cloud auto-scaling with SLA objective using Q-learning. In: Proc. of IEEE FiCloud '18. pp. 85–92 (2018)
8. Jawarneh, I.M.A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., Palopoli, A.: Container orchestration engines: A thorough functional and performance comparison. In: Proc. of IEEE ICC 2019. pp. 1–6 (2019)
9. Jerzak, Z., Ziekow, H.: The DEBS 2015 grand challenge. In: Proc. ACM DEBS 2015. pp. 266–268 (2015)
10. Khazaei, H., Ravichandiran, R., Park, B., Bannazadeh, H., Tizghadam, A., Leon-Garcia, A.: Elascale: Autoscaling and monitoring as a service. In: Proc. of CASCON '17. pp. 234–240 (2017)
11. Mao, Y., Oak, J., Pompili, A., Beer, D., Han, T., Hu, P.: DRAPS: dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster. In: Proc. of IEEE IPCCC '17. pp. 1–8 (2017)
12. Nardelli, M., Cardellini, V., Casalicchio, E.: Multi-level elastic deployment of containerized applications in geo-distributed environments. In: Proc. of IEEE FiCloud '18. pp. 1–8 (2018)
13. Rossi, F., Cardellini, V., Lo Presti, F.: Elastic deployment of software containers in geo-distributed computing environments. In: Proc. of IEEE ISCC '19. pp. 1–7 (2019)
14. Rossi, F., Nardelli, M., Cardellini, V.: Horizontal and vertical scaling of container-based applications using Reinforcement Learning. In: Proc. of IEEE CLOUD '19. pp. 329–338 (2019)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 2 edn. (2018)
16. Tang, Z., Zhou, X., Zhang, F., Jia, W., Zhao, W.: Migration modeling and learning algorithms for containers in fog computing. IEEE Trans. Serv. Comput. **12**(5), 712–725 (2019)
17. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A hybrid Reinforcement Learning approach to autonomic resource allocation. In: Proc. of IEEE ICAC '06. pp. 65–73 (2006)
18. Zhao, D., Mohamed, M., Ludwig, H.: Locality-aware scheduling for containers in cloud computing. IEEE Trans. Cloud Comput. (2018)