A Service-Aware Autoscaling Strategy for Container Orchestration Platforms with Soft Resource Isolation

Citation for the original published paper (version of record):
Tonini, F., Natalino Da Silva, C., Temesgene, D. et al (2023). A Service-Aware Autoscaling Strategy for Container Orchestration Platforms with Soft Resource
Isolation. 2023 JOINT EUROPEAN CONFERENCE ON NETWORKS AND COMMUNICATIONS & 6G SUMMIT, EUCNC/6G SUMMIT: 454-459.
http://dx.doi.org/10.1109/EUCNC/6GSUMMIT58263.2023.10188268

A Service-Aware Autoscaling Strategy for Container Orchestration Platforms with Soft Resource Isolation

Federico Tonini
and Carlos Natalino
Electrical Engineering Department
Chalmers University of Technology
Gothenburg, Sweden

Dagnachew A. Temesgene
and Zere Ghebretensaé
Ericsson Research
Kista, Stockholm, Sweden

Lena Wosinska
and Paolo Monti
Electrical Engineering Department
Chalmers University of Technology
Gothenburg, Sweden

***Abstract*—Container orchestration platforms like Kubernetes (K8s) allow easy deployment and management of cloud native services. When deploying their services, service providers need to specify a proper amount of resources to K8s, so that the desired Quality of Service (QoS) to their users can be maintained. To cope with the varying traffic demand coming from users, they can rely on the K8s Horizontal Pod Autoscaling (HPA) mechanism. To ensure that enough resources are available when needed, the standard HPA mechanism relies on resource overprovisioning. In this way, the required QoS is achieved most of (or all) the time but at the expense of additional resources that are allocated (and charged for), while they may stay idle for significant periods of time. A way to reduce overprovisioning is provided by the soft resource isolation of K8s, which allows services to compensate for a temporary lack of resources with shared resources, i.e., idle resources of the machines where services are running. However, during traffic spikes, these idle resources may not be enough to serve the whole demand, degrading the QoS. The HPA, which is not aware of how much demand could not be served, is not always able to correctly estimate the required additional resources, further degrading the QoS. To overcome this, service providers need to leverage overprovisioning, limiting the use of shared resources. In this paper, we propose a novel mechanism for autoscaling resources in K8s that relies on service-related data to avoid the additional degradation introduced by the HPA. The proposed strategy also offers a way to tune overprovisioning and shared resources. Simulation results show that our approach can reduce idle resources by up to 60% compared with the HPA mechanism.**

***Keywords*—Cloud native services, QoS, service degradation, Pod autoscaling, Kubernetes, Shared resources, Pod as a Service.**

I. Introduction

The advent of containers and related orchestration platforms enabled the possibility to develop and deploy cloud native services [1]. Kubernetes (K8s), an example of a widely used platform, offers easy and automatic container deployment and management [2]. Thanks to K8s, service providers can rent collections of containers, called Pods, from cloud providers in a *Pod-as-a-Service* fashion. Service providers usually specify the amount of resources (e.g., memory and CPU) assigned to each Pod so that a required Quality of Service (QoS) is offered to the users. If the resources are insufficient to satisfy the user demand over time, users may experience QoS degradation, e.g., in the form of an increased application response time. This, in turn, may result in a loss of revenue for service providers [3]. A common way to alleviate this is to overprovision resources/Pods, which increases costs for service providers who need to pay for resources even during the time when they are not used (idle).

K8s provides a scaling mechanism called Horizontal Pod Autoscaling (HPA). The HPA reads the current CPU load over the running replicas and scales the Pods when a threshold is exceeded, in a reactive way [4]. However, the process of scaling is not instantaneous as it takes some time (usually referred to as scaling delay) to create replicas, update the necessary components, and set up the service(s) within the Pod. During this time, resources may not be enough to cope with traffic variations, causing degradation, usually compensated with overprovisioning. A possible alternative to reduce degradation without relying too much on overprovisioning is to leverage the soft isolation provided by K8s. Thanks to soft isolation, resources can be shared among other Pods that run on the same machine (either physical or virtual). In this way, a service/Pod can temporarily use idle resources of another service/Pod to compensate for a lack of resources, mitigating degradation. During service deployment, a service provider can specify the amount of dedicated and shared resources the Pods can access. In [5], we leverage the HPA mechanism to show that using shared resources allows service providers to reduce the number of required CPUs, leading to significant cost savings. Even though this approach provides benefits, the HPA relies on a scaling mechanism that is not designed to work with shared resources. In fact, when a service relies on shared resources to cope with spikes in the demand, it is possible that part of the needed resources is not available, causing degradation. In this situation, the HPA will scale the Pods to cope with the lack of resources but at a rate that might be too slow, causing additional degradation. This is because the scaling decisions made by the HPA are based not on the total number of resources needed to accommodate the extra demand, but only on whether or not the resources used by Pods exceed the value set by the threshold.

Different works tackle the HPA inefficiencies by leveraging Artificial Intelligence (AI) and Machine Learning (ML) techniques to learn characteristics of the system and adjust the number of replicas accordingly [6]–[11]. However, all these works rely on dedicated resources and overprovisioning. They do not consider the possibility to leverage soft isolation and, therefore, do not provide a way to balance shared and dedicated resources.

In this paper, we propose a novel Service Aware Pod Autoscaling (SAPA) mechanism that leverages service-related

information (e.g., user requests) to calculate the desired number of Pods. Conversely to the HPA, this allows also taking into account the unserved demand and scaling accordingly to the demand variations even when some of the traffic cannot be served. In addition, this mechanism offers the possibility to control the balance between resource overprovisioning and shared resources by tuning a parameter. Simulation results show that the SAPA mechanism is able to reduce up to 60% of the idle resources compared to the K8s HPA mechanism with shared resources while not introducing significant degradation.

II. Related Work

In the literature, different solutions have been proposed to improve the HPA mechanism. The authors in [6] predict the load of the Pods in advance and adjust the number of replicas accordingly. The paper in [7] proposed a reinforcement learning approach to automate and guide the scaling based on online and continuous learning of the system. The works in [8]–[11] adopts service-related metrics to improve the HPA. In [8], the authors use service latency as a metric to drive the scaling process, instead of CPU usage. The authors in [9] proposed a proactive scaling procedure based on burst predictions to satisfy QoS requirements while optimizing resource utilization for containers in K8s. In [10], the authors propose a proactive scaling engine that leverages user demand predictions. The authors in [11] operate microservices with different objectives such as end-to-end delay bounds on service requests, throughput, and service differentiation. The resources are prioritized among the microservices to ensure the performance objectives.

All these works represent significant advancements with respect to the HPA as resources are used more efficiently. However, the proposed solutions still rely on resource overprovisioning to compensate for sudden traffic spikes and/or prediction inaccuracies. All the works focus on the case of services that run on dedicated resources, i.e., resources that the service provider pays for regardless of how much they are used. Such resources can be accessed at any time during the service operation. With soft isolation, instead, service providers are required to decide not only on the amount of dedicated resources but also on the number of shared resources to rely on. The existing strategies are, therefore, not suitable for this case. The HPA could be used, but it does not provide an efficient way to scale with shared resources. Therefore, a novel scaling strategy is needed to alleviate the issues of the HPA in the presence of shared resources.

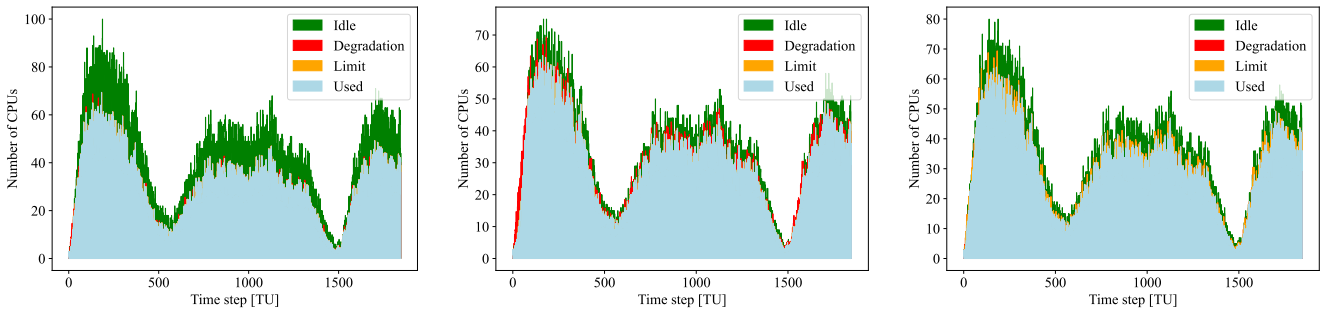
III. Autoscaling with Shared Resources

Cloud resources (e.g., CPUs and memory) are assigned by K8s to Pods/services in two different ways: dedicated and shared [12]. Dedicated resources are guaranteed, i.e., they are assigned to Pods/services and paid for regardless of their use, and can be accessed by the Pods/service at any time during the service operation. Conversely, shared resources are assigned on a best-effort basis depending on their availability, and are paid for only when used. In the following, we will refer to dedicated and shared resources as *request* and *limit* resources, respectively. The service provider, who rents compute and memory resources from the cloud provider, can specify how many dedicated and shared resources each Pod can use by means of resource request and limit of K8s, respectively [12]. The number of Pods is automatically adjusted by K8s over time via an HPA mechanism. It creates or terminates Pod replicas to cope with service demand variations, allowing service providers to pay only for the resources needed to operate their services. The HPA mechanism periodically monitors the average CPU load among the Pods and compares this value with a scaling threshold, i.e., the amount of CPUs that triggers the creation of new Pod replicas if exceeded. Thus, the scaling threshold also represents the maximum desired average CPU load across all Pods of a service. The desired number of replicas (des. #replicas) is calculated based on the current number of replicas (cur. #replicas), the current metric value (cur. metric value), and the desired metric value (des. metric value), according to the following formula [4]:

$$\text{des. \#replicas} = \left\lceil \text{cur. \#replicas} \cdot \frac{\text{cur. metric value}}{\text{des. metric value}} \right\rceil . \quad (1)$$

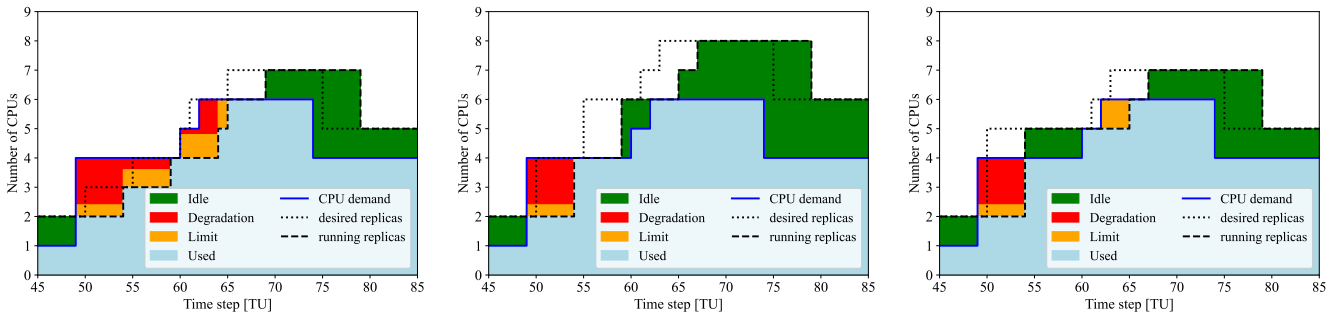
In this work, we consider CPU usage as the metric driving the scaling process. In this case, the current metric value is the average CPU usage over all active Pods and the desired metric value corresponds to the scaling threshold. In K8s, the threshold is expressed as a percentage of the request and is converted into the corresponding CPU amount in (1).

The scaling process takes some time, referred to as scaling delay, needed to instantiate the new Pod(s) and to start all the included service components. During this process, degradation is experienced if the allocated resources are not enough to satisfy the demand. Degradation is therefore a lack of CPUs to serve the demand, lowering the QoS for the users. A service provider can decide to rely only on request resources to provide its services. In this case, the service provider can tune the scaling threshold to control the degradation by means of resource overprovisioning. If the scaling threshold is low (e.g., 70%), the HPA keeps the average CPU load per Pod equal to this value. Hence, a lot of request resources are idle (e.g., 30% of the resources are idle). This case is shown in Fig. 1a, where an example of service deployment with Pods with 1 request CPU and a threshold set to 70% is depicted. In particular, the figure shows the request resources that are either used or idle, and the degradation in [CPUxTU]. The Pods are not allowed to use limit resources in this case. To reduce idle request resources, the service provider can decide to set a higher scaling threshold. However, an excessive increase in the threshold may lead to degradation as the resources may not be enough to satisfy changes in user demand. This is shown in Fig. 1b, where the scaling threshold has been increased to 85%, and we can notice a substantial degradation when the user traffic fluctuates. An alternative for service providers is to use limit resources, instead of overprovisioning, to compensate for the degradation. In this case, idle resources of other Pods running on the same machine, or idle resources of the machine itself, can be used to temporarily increase the Pod resources. By doing so, the service provider can keep a high scaling threshold, with limited overprovisioning, and rely on limit resources to compensate for the degradation, betting on the fact that those resources will be available when needed. An example of this is depicted in Fig. 1c for a service that can leverage limit resources with a scaling threshold set to 85%. Compared to Fig. 1a, the degradation is similar but with a



(a) Request resources only, scaling threshold 70%. (b) Request resources only, scaling threshold 85%. (c) With limit resources, scaling threshold 85%.

Fig. 1. Example of a cloud native service operated using K8s without and with limit resources. The used, idle, degradation, and limit resources for three different settings are reported. Time is discretized and expressed in Time Units (TUs).



(a) HPA with limited overprovisioning. (b) HPA with large overprovisioning. (c) SAPA with limited overprovisioning.

Fig. 2. Sample case showing the result of operating a cloud native service using K8s HPA and SAPA mechanism. Time is discretized and expressed in Time Units (TUs).

much lower amount of idle resources. Compared to Fig. 1b, the amount of idle resources is similar but the degradation is compensated by limit resources.

Relying on the scaling mechanism of the HPA to control shared resources is not efficient. In fact, to be able to use limit resources, a service provider must set a high scaling threshold. In this situation, there is a risk that part of the limit resources required to serve the demand is not available, causing degradation. The HPA mechanism based on (1) relies only on the information on the load and is not aware of how much demand could not be served. Consequently, the number of desired replicas computed with (1) may be underestimated. An example of this is represented in Fig. 2a, with the HPA that is set to work with a scaling threshold of 91% and Pods with 1 request CPU. It is possible to notice that, after the demand increases (at 49 [TU]), the HPA computes (with (1)) a desired number of replicas that is lower than what is actually needed. During the scaling delay set to 4 [TU] in this example, the system is able to partially compensate for the lack of resources with limit resources, and some degradation is experienced. At time instant 54 [TU] the number of running replicas is adjusted, but still not enough to satisfy the demand. The HPA computes a new desired number of replicas, which is achieved after the scaling delay at time 59 [TU]. During this time, additional degradation is experienced. To cope with this issue, the service provider must lower the scaling threshold, increasing the overprovisioning. An example is reported in Fig. 2b where the threshold is lowered to 75% for the same CPU demand. It can be seen that, due to the lower scaling threshold, the number of desired replicas obtained with (1) is higher than in the previous example (in Fig. 2a). Degradation is lower and is experienced only during the scaling delay between time 49 [TU] and 54 [TU]. Fewer limit resources are used and the idle is higher, due to the large overprovisioning. This example shows that with the HPA it is difficult to leverage limit resources without increasing the degradation. To solve this issue, we propose a Service Aware Pod Autoscaling (SAPA) mechanism that makes the scaling system aware of the entity of the unserved demand to better estimate the desired replicas. An example of the outcome of this strategy is reported in Fig. 2c for the same CPU demand. In this case, the scaler at time 49 [TU] is aware of the entity of the demand and adjusts the number of replicas accordingly, after the scaling delay (time 54 [TU]). Compared to the case in Fig. 2a, the service experiences less degradation with a similar idle. Compared to Fig. 2b, the service experience the same degradation while leveraging more limit resources and lowering the idle. In the next section, we describe in detail the SAPA architecture and scaling mechanism.

IV. The Service Aware Pod Autoscaling (SAPA) Mechanism

This section introduces the novel scaling mechanism named SAPA that provides the means to control the amount of limit

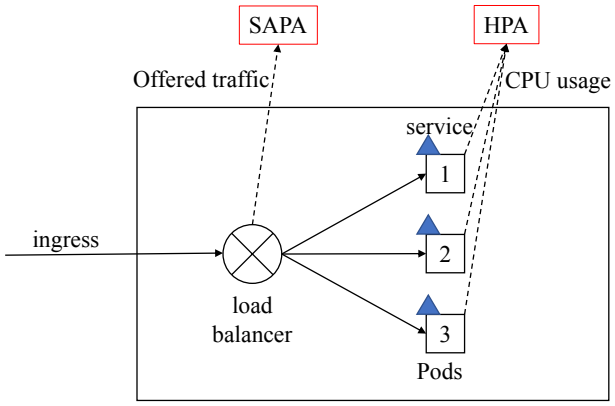


Fig. 3. The proposed SAPA architecture.

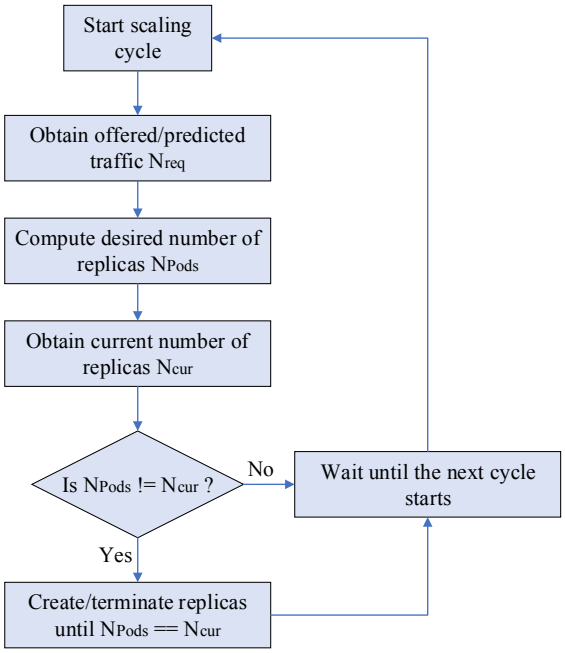


Fig. 4. The flowchart of the proposed SAPA mechanism.

resources to be used by relying on service-related information. More specifically, we assume that the service provider knows how many user requests per unit of time each Pod can process using request resources without degrading the QoS. Figure 3 shows the architecture of the proposed SAPA mechanism. Service requests enter the system and arrive at a load balancer, which is a software entity in charge of forwarding requests to different Pod replicas. Service requests are then processed by the Pods. While the HPA relies on CPU load measurements, the SAPA relies on data from the load balancer. This allows for more accurate computation of the desired number of replicas by considering the total traffic, i.e., both the served and unserved service demands.

Figure 4 reports the main steps of the SAPA mechanism. An autoscaler evaluates periodically the need for scaling. First, the offered traffic in terms of the number of requests (N_{req}) is obtained. This value can be either the current value or a prediction of the service requests that are expected after a number of TUs. In the latter case, the prediction can be used

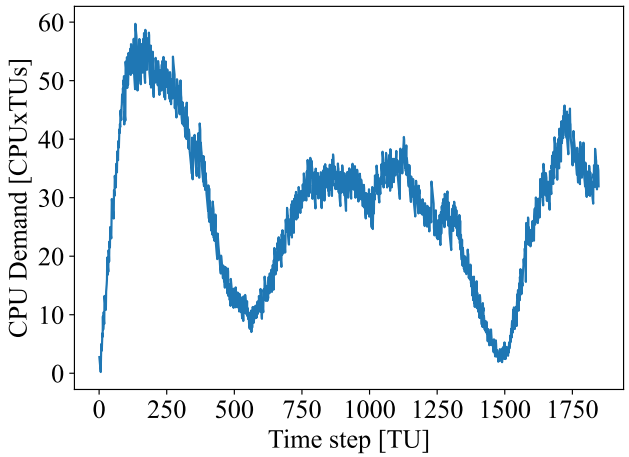


Fig. 5. CPU demand over time [7].

to reduce degradation during the scaling delay, as the system is able to start the scaling procedure in advance. The specific prediction strategy to be adopted is out of the scope of this paper, because only the value of N_{req} is needed by SAPA. The number of desired Pod replicas (N_{Pods}) is computed based on the following formula:

$$N_{Pods} = \left\lceil \alpha \cdot \frac{N_{req}}{M_{req}} \right\rceil, \tag{2}$$

where α is a parameter to allow for underdimensioning/overdimensioning of the Pod replicas and M_{req} is the maximum supported request rate per Pod without degrading the QoS. N_{req}/M_{req} provides the number of replicas that are required to satisfy the demand with the Pod request. By tuning the value of α, the value of N_{Pods} can be changed allowing overloading or underloading of the Pods. This, in turn, changes the share between the Pod request and limit to be used.

After N_{Pods} is computed, the current number of Pod replicas (N_{cur}) is obtained. N_{Pods} and N_{cur} are then compared and, if they are not equal, a number of Pod replicas are created/terminated to increase/decrease the N_{cur} up/down to the value of N_{Pods}. Then, the system waits until the next scaling cycle.

V. Numerical Results

To prove the benefits of the proposed approach, we performed simulations showing the benefits of SAPA against a conventional scaling procedure leveraging HPA. We developed a Python-based custom simulator that mimics Kubernetes monitoring, load balancing, and scaling behavior. Periodically (with a period of 1 TU) we evaluate the demand of each service deployed and compute the number of desired replicas applying three approaches: HPA, SAPA using instant demand to drive the scaling (without predictions), and SAPA with predicted service requests. In the following, we show results where resources are measured in [CPU×TUs].

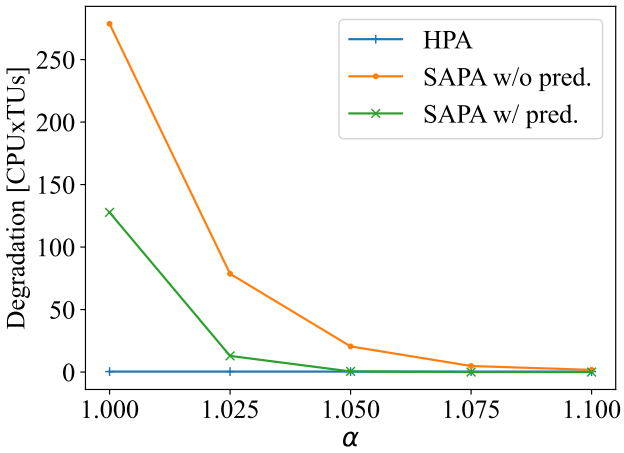


Fig. 6. Degradation (in [CPUxTU]) as a function of different values of the scaling coefficient α for HPA, SAPA without predictions (SAPA w/o pred.), and SAPA with predictions (SAPA w/ pred.).

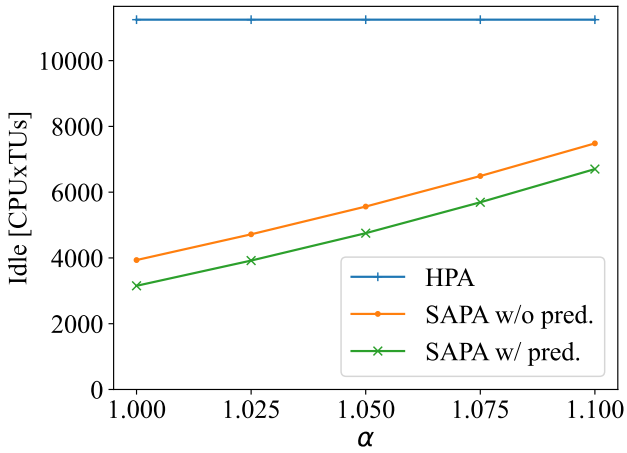


Fig. 7. Idle resources (in [CPUxTU]) as a function of different values of the scaling coefficient α for HPA, SAPA without predictions (SAPA w/o pred.), and SAPA with predictions (SAPA w/ pred.).

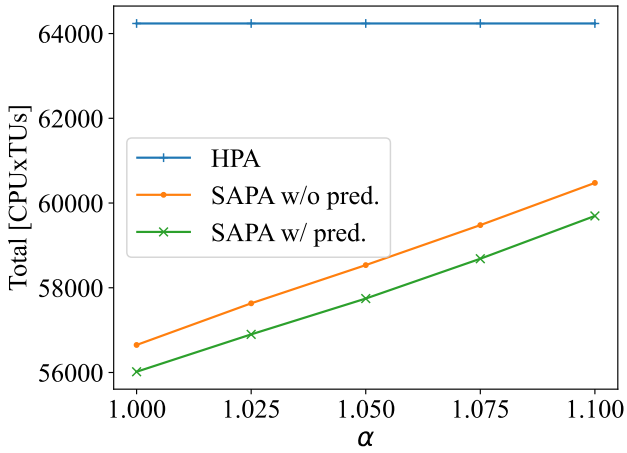


Fig. 8. Total resources (in [CPUxTU]) as a function of different values of the scaling coefficient α for HPA, SAPA without predictions (SAPA w/o pred.), and SAPA with predictions (SAPA w/ pred.).

A. *Simulation settings*

We consider 50 services with a CPU demand over time according to the CPU demand pattern shown in Fig. 5 [7]. We assume that $M_{req} = 1$ and that the CPU demand in Fig. 5 is equivalent to N_{req}, to have a fair comparison with the HPA. The different service traces are obtained by shifting the demand pattern over time. At each time step, we add a random uniform value in the interval of $\pm 20\%$ to each sample of the workload to mimic user traffic variations. For each service, we divide the CPU demand equally among the running replicas, simulating a perfect load-balancing scheme. The scaling delay and time for prediction are set to 4 TUs to account for the time needed to create/terminate Pods, update load-balancing components, and set up/terminate the services. The obtained results are an average of 10 simulations with a confidence interval always within $\pm 1\%$ with a confidence level of 95%. For each Pod, we assume that the Pod request CPU is 1 while the Pod limit is set to a large value, so that each Pod can access up to all unused (or idle) CPUs on the machine in which they are deployed. We consider a cluster with machines of 24 CPUs each. The HPA is set to scale with a threshold of 85%, which is the largest value that guarantees that the degradation is not significant (e.g., below 0.001%) and is considered to be the benchmark. The SAPA with prediction is fed by the original workload before applying the random traffic variation, which mimics inaccuracy in the prediction process.

B. *Simulation results*

Figure 6 depicts the total degradation (i.e., the amount of CPUs that was required by user demand but could not be provided) for different values of α for the three different strategies, averaged over the services. It is possible to observe that HPA with 85% threshold does not introduce significant degradation. On the other hand, the SAPA degradation depends on the specific setting of α. The SAPA without predictions (denoted as SAPA w/o pred.) is able to obtain the same degradation as HPA when $\alpha = 1.1$. The SAPA with predictions (denoted as SAPA w/ pred.), even though is subject to inaccuracies, performs better than the SAPA without predictions. More specifically, degradation matches the one provided by HPA when $\alpha = 1.05$.

Figure 7 reports the number of idle CPUs for the HPA, SAPA without predictions, and SAPA with predictions, averaged over the services. This metric directly relates to resources that were paid for but remained unused. From the figure, it is possible to notice that the idle resources increase with α, as increasing α makes the number of desired Pods larger, hence leading to overprovisioning. Let us compare the cases where the three approaches do not provide significant degradation. The SAPA with predictions (at $\alpha = 1.05$) achieves a reduction of almost 60% idle resources compared to the HPA, while the SAPA without predictions (at $\alpha = 1.1$) achieves a reduction of around 30%. This confirms that, by selecting a proper value for α, the proposed approach can achieve the targeted degradation level (or QoS) while reducing idle resources

compared to the overprovisioning in a conventional strategy based on HPA. Reducing the overprovisioning in HPA would require increasing the scaling threshold. However, it would also increase the probability that the demand cannot be satisfied, hence it would lead to higher degradation. This is due to the fact that if degradation is experienced, the K8s scaling formula (1) is not always able to correctly estimate the desired number of replicas. This error propagates over time as the incorrect desired number of replicas becomes the new current number of replicas in the subsequent calculation.

Figure 8 shows the average amount of total (i.e., the sum of used, idle, and limit) resources per service, needed by the three approaches. As for the idle resources, also the total resources depend on the value of α. Moreover, the HPA is always outperformed by the SAPA. More specifically, the SAPA with predictions and $\alpha = 1.05$ requires the least amount of resources to achieve the same low level of degradation as the HPA, followed by the SAPA without prediction and $\alpha = 1.1$. In this case, the SAPA without and with predictions requires 6% and 11% fewer resources compared to the HPA, respectively, to run a service.

VI. Conclusion

In this paper, we focus on the trade-off between resource overprovisioning and degradation from a service provider perspective in cloud native services based on K8s. In particular, we propose the novel SAPA mechanism that relies on limit resources and service-related information from the load balancer to optimize resource usage. In contrast to the K8s HPA, which relies on the average CPU usage, the proposed mechanism is able to account for the unserved service demand during the scaling process and provides a simple way to tune the amount of limit resources. This results in more efficient use of resources, reflected in reduced overprovisioning. Simulation results show that the proposed SAPA mechanism offers much lower degradation than the HPA, as well as reducing idle resources by around 60% and 30% in the case with and without user request predictions, respectively. Although the proposed approach represents a substantial step towards eliminating idle resources, further studies are necessary to eliminate idle resources completely. In future work, we are planning to evaluate the proposed mechanism in a real setup.

Acknowledgment

This work was supported by EUREKA cluster CELTIC-NEXT projects AI-NET-ANIARA and AI-NET-PROTECT funded by VINNOVA.

References

[1] "Oracle - What is Cloud Native?" https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/.

[2] "Kubernetes," https://kubernetes.io/.

[3] Akamai, "The state of online retail performance," Tech. Rep., 2017. [Online]. Available: https://s3.amazonaws.com/sofist-marketing/State+of+Online+Retail+Performance+Spring+2017+-+Akamai+and+SOASTA+2017.pdf

[4] "Kubernetes Horizontal Pod Autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[5] F. Tonini, C. Natalino, D. A. Temesgene, Z. Ghebretensaé, L. Wosinska, and P. Monti, "Benefits of pod dimensioning with best-effort resources in bare metal cloud native deployments," *IEEE Networking Letters*, vol. 5, no. 1, pp. 41–45, 2023.

[6] A. Zhao, Q. Huang, Y. Huang, L. Zou, Z. Chen, and J. Song, "Research on resource prediction model based on Kubernetes container auto-scaling technology," *IOP Conference Series: Materials Science and Engineering*, vol. 569, no. 5, p. 052092, jul 2019.

[7] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2019, pp. 329–338.

[8] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *IEEE International Conference on Web Services (ICWS)*, 2019, pp. 68–75.

[9] D.-D. Vu, M.-N. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 109 768–109 778, 2022.

[10] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive AI-based auto-scaling for Kubernetes," in *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 599–608.

[11] F. Shahab Samani and R. Stadler, "Dynamically meeting performance objectives for multiple services on a service mesh," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 219–225.

[12] "Kubernetes Resources," https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.