High-Level Metrics for Service Level Objective-aware Autoscaling in Polaris: a Performance Evaluation

Nicolò Bartelucci, Paolo Bellavista
Department of Computer Science and Engineering
University of Bologna
Bologna, Italy
nicolo.bartelucci@studio.unibo.it, paolo.bellavista@unibo.it

Thomas Pusztai, Andrea Morichetta, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Vienna, Austria
(t.pusztai, a.morichetta, dustdar)@dsg.tuwien.ac.at

2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC) | 978-1-6654-9524-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICFEC54809.2022.00017

***Abstract*—With the increasing complexity, requirements, and variability of cloud services, it is not always easy to find the right static/dynamic thresholds for the optimal configuration of low-level metrics for autoscaling resource management decisions. A Service Level Objective (SLO) is a high-level commitment to maintaining a specific state of a service in a given period, within a Service Level Agreement (SLA): the goal is to respect a given metric, like uptime or response time within given time or accuracy constraints. In this paper, we show the advantages and present the progress of an original SLO-aware autoscaler for the Polaris framework. In addition, the paper contributes to the literature in the field by proposing novel experimental results comparing the Polaris autoscaling performance, based on high-level latency SLO, and the performance of a low-level average CPU-based SLO, implemented by the Kubernetes Horizontal Pod Autoscaler.**

***Index Terms*—Cloud, Edge, Computing, Autoscaling, Polaris, Kubernetes, Performance, Evaluation, Horizontal, Pod, Autoscaler, Elasticity, High-level, SLO, Horizontal Pod Autoscaler**

I. Introduction

With the recent and rapid diffusion of edge and cloud computing technologies, virtualization techniques have become the state-of-the-art of big ICT infrastructures, by enabling high availability, high scalability, and high elasticity. Elasticity, i.e., the ability to dynamically grow or shrink infrastructure resources to autonomously adapt to workload changes, can be enabled with three main approaches: vertical (increase the system resources), horizontal (increase the number of working replicas), or their combination [1]. In this context, containerization techniques are the most widespread due to their lightweight and portable solutions. One of the most performing related orchestration tool is Kubernetes [2], which has demonstrated to achieve good scalability and availability. Kubernetes comes with different autoscaling add-ons that enable elasticity among pods; among them, the most known and utilized is the Horizontal Pod Autoscaler (HPA), which comes out of the box with Kubernetes deployment [3] and can monitor a given metric (like CPU usage) and consequently scale-out the deployed system when the specified metric overcomes a static threshold.

A Service Level Objective (SLO) [4] allows providers to define expected performance in the given products and allows consumers to know what to expect from the service itself. The SLO is a "commitment to maintaining a particular state of the service in a given period" [5] . Usually it relies on business-relevant Key Performance Indicators (KPIs) or Service Level Indicators (SLI). Tools like HPA only offer basic support for SLOs and this means that customers must adapt their SLO to measurable low-level metrics like CPU or memory usage. Instead, new and more SLO-oriented frameworks for QoS management, such as Polaris [6], allow to define flexible scaling strategies, both horizontal and vertical, and to apply them by monitoring both low-level and high-level metrics, whose monitoring information is gathered via widespread and state-of-the-art monitoring infrastructures such as Prometheus.

In this short paper, we first present the advantages of high-level metrics for SLO-aware autoscaling, and then we thoroughly evaluate the performance of Polaris as a high-level metric-based pods autoscaler for Kubernetes. To this purpose, we compare it to HPA, by showing how our proposed approach can make more reactive decisions as the load changes and can provide better service quality from the user perspective. In particular, Section 2 motivates the need of high-level metrics-based autoscaling mechanisms; Section 3 compares our original approach in Polaris with state-of-the-art solutions currently available in the literature; Section 4 introduces novel SLOs for latency; and finally Section 5 reports about the performance results achieved by our proposal vs. HPA based on real in-the-field experimentation.

II. Problem Statement and Motivations

In cloud computing, it is common for providers and consumers to agree on SLAs to define bounds within which a certain cloud service has to operate [7]. The SLA is made of one or more SLOs and allows to define high-level indicators seen as a value or state to be maintained during the service provisioning period. Since SLOs are guarantees given to the customer, respecting SLOs is considered a crucial positive feature for cloud providers. However, it is also important, from

978-1-6654-9524-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ICFEC54809.2022.00017

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on June 27,2022 at 06:55:32 UTC from IEEE Xplore. Restrictions apply.

a service provider perspective, not to over-provision resource allocation in a deployment environment while considering a given SLO, as it would result in additional costs and non-optimal resource utilization.

Autoscaling technologies allow to minimize the over- or under- provisioning issues, by dynamically adapting the enforced/deployed configuration to the current specific requirements and situation. However, in most solutions, like the Kubernetes HPA, they only allow to map the elasticity of a cloud application to a low-level metric (i.e., a metric considering low-level monitoring indicators such as CPU usage) and it is hard for a service provider to control the agreed high-level SLO [8]. Polaris overcomes this problem and allows cloud providers to directly map their SLOs to higher level metrics. As a result, elastic horizontal scaling strategies based on high-level performance indicators such as latency have proven us to be well-suited for highly-dynamic and interactive applications, where our SLO is to maintain the agreed response time; this will be also confirmed from the novel results later reported in this paper (see Section V). We consider latency a high-level metric because it can be perceived directly by the end user and is not based only on the properties of the host machine but also on the infrastructure ones. To develop and create additional controllers, Polaris comes with SLO Script [9]: a language and framework for developing complex SLOs and elasticity strategies, based on one or more metrics that are configurable and composable via the aggregation of different metrics.

III. Related Work

This section explores the state of the art in terms of cloud computing elasticity and its application. Al-Dhuraibi et al. [10] released a survey discussing the state of the art of elasticity in cloud computing, by focusing their work on VMs and container-based elasticity. They present the various shapes of elasticity in reference to cloud computing as policies, methods, architectures, configurations, scopes, and enabling technologies. In their work, they set a research challenge, the "Thresholds definition": elasticity strategies are based on thresholds for the measured metrics such as CPU or memory utilization. However, they claim that choosing a suitable threshold is not an easy task due to the changes in workload, application behavior, and the possible combination of the two.

Many commercial tools for elasticity in cloud computing are currently present in the market. Most of them are from big commercial cloud providers as AWS [11], Azure [12], and Google [13]. They provide solutions for simple SLOs, plus they are tied to the use of specific provider services. Others are out-of-the-box solutions like HPA [3], Vertical Pod Autoscaler [14] (VPA), and Cluster Autoscaler [15] (CA) that come with the standard Kubernetes deployment [2]. These solutions only allow, by default, to realize runtime resource scaling based on low-level metrics, like CPU or memory usage. To leverage custom metrics, they require setting up other custom components. On the contrary, Polaris [6] is based on elasticity strategies that can be vertical, horizontal, or custom. Polaris realizes full decoupling between the elasticity strategy and the SLO controller: these two components are connected by the the user configuration, which is called SLO Mapping [9]. Polaris can integrate with Prometheus for gathering monitoring indicators for its metrics and has a smart control loop to apply scale in or scale out of replicas. Polaris SLOs can be reconfigured by changing the given SLO Mapping without changing the configuration of the deployed application-level components.

IV. High-level Latency-based SLO

In this section we present the policies and the middleware components that we have implemented, deployed, and run in order to execute our performance comparison between Polaris and HPA, with the ultimate goal to verify and demonstrate how scaling based on high-level metrics is feasible and can achieve better results.

As shown in [6], the Polaris SLO Control Loop needs two main entities. Firstly an SLO Controller, that evaluates the SLO and computes an SLO Compliance value. Secondly, an Elasticity Strategy Controller, that includes the policy to scale the replicas. In our tests, we use a Horizontal Elasticity Strategy Controller. Considering R_{new} as the new replica number, R_{old} as the old replica number and k as the SLO Compliance value, we compute the new pods replica number according to (1). The SLO Compliance value tells how much we are currently respecting the SLO.

$$R_{new} = \lceil R_{old} * \frac{k}{100} \rceil \quad (1)$$

For example, when it has a value of 100, it means that the provider perfectly fulfills the SLO. The output of the SLO Controller, in our case a Latency SLO Controller, utilizes the current and the target latency values for the calculation. Given L_q the n-th percentile of the latency values in a given time window, and $\theta_{latency}$ the latency threshold that violates the SLO, we compute the SLO Compliance value k according to (2).

$$k = \lfloor \frac{L_q}{\theta_{latency}} * 100 \rfloor \quad (2)$$

The Polaris solution for autoscaling provides a decoupled way to create elastic services and the decoupling is enabled by the generic SLO Compliance value that is not tied to a particular SLO type. To this purpose, we developed the Latency SLO Controller and the Elasticity Strategy Controller from scratch with the SLO Script framework from Polaris. Furthermore, we developed support for histogram quantile metric in SLO Script to compute the percentiles for latency.

In addition, HPA is based on monitoring and scaling by considering the average CPU utilization, due to its lack of support for high-level SLOs. There is no decoupling between what and when a strategy has to be applied. The number of replicas to deploy is computed according to equation (3), as stated in the official Kubernetes documentation [3].

$$R_{new} = \lceil R_{old} * \frac{M_t}{\theta_{cpuusage}} \rceil \quad (3)$$

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on June 27,2022 at 06:55:32 UTC from IEEE Xplore. Restrictions apply.

where M_t is the measured average CPU utilization in the instant t and $\theta_{cpuusage}$ is the target CPU usage.

V. Evaluation

In order to show the benefits of using a high-level based SLO scaling, we leverage a private cloud infrastructure to deploy a Tensorserving inference service, with HPA and Polaris managing its scaling for our benchmarking purposes. Specifically, we are interested in showing how the Polaris solution, based on high-level SLOs, can decrease latency and be more reactive in scaling the system out, while still providing good service quality and reducing costs even when the system would tend to be over-provisioned according to the SLO values.

A. Setup

In our experimentation work, we deployed a Kubernetes cluster (Microk8s) composed by two nodes hosted in different VMs with 24 vCPUs and 48 Gb of memory each. The Kubernetes version used is 1.21.1 of both Kubernetes Client and Server. The machines hosting the nodes are interconnected with a gigabit Ethernet LAN. We deploy Polaris, HPA, Prometheus, Grafana Dashboard, and Ingress-NGINX in our Kubernetes Cluster. Prometheus, a dependency of Polaris, has been used to gather metrics to let the SLO Controller compute the SLO Compliance value. Grafana, an open-source analytics and monitoring solution for databases, was used to evaluate the metrics after the test runs. Ingress-NGINX, an Ingress controller for Kubernetes using NGINX as a reverse proxy and load balancer, creates a single ingress for the Tensorserving pods and gathers the metrics about their latency. We deploy Polaris, Grafana, and Ingress-NGINX with a helm chart with the limit to use 100 milli-CPUs, the default value proposed by the official documentations. The Tensorserving system is deployed as a single pod, to be scaled by Polaris or HPA. Tensorserving pods are limited to use 2 vCPUs maximum. The model for inference is a MobileNetV2 [16] that proved to be very well suited to create high computation load in the Kubernetes cluster.

B. Metrics

The metrics that have been used in the following experiments for performance evaluation are shown in the code excerpts of Figures 1 and 2.

```
histogram_quantile(
  <quantile_value>,
  sum by (le)(
    rate(
      nginx_ingress_controller_request_duration_seconds_bucket{
        ingress =~ "$ingress"
      }[1m]
    )
  )
)
```

Listing 1. Latency n-th Percentile

The PromQL query in Listing 1 has been used to evaluate latency. In particular. it uses percentiles of values and the network speed is neglected in these experiments as the worker clients are placed in the same location of the virtual machines (the latency is calculated only at the Ingress-NGINX level). The percentiles that we use for the evaluation are 90th and 95th, as regularly used by commercial cloud providers to measure metrics, like Amazon AWS does [17]. The request volume in Listing 2 has also been considered to monitor the network load that changes during provisioning.

```
sum by (path)(
  rate(
    nginx_ingress_controller_request_duration_seconds_count{
      ingress =~ "$ingress"
    }[1m]
  )
)
```

Listing 2. Request Volume

It is useful for the analysis to see how the load changes in the network. Median latency (50th percentile latency) and replica number have also been used for our evaluation.

C. Experiments

To prove how high-level metric-based scaling can be more effective in maintaining a given SLO, we run a wide and extensive set of stress tests. In the first set of tests, Polaris SLO is set to 500ms latency on a 90th request percentile and is compared with HPA set to 70% of CPU utilization. In the second set of tests, the Polaris SLO is set to 600ms latency on a 95th request percentile and is compared with HPA set to 50% of CPU utilization. These CPU usage and latency values were determined using the latency of an inference request made while the overall infrastructure in the employed deployment environment was idle. Both autoscaling systems have a stabilization window of 45 seconds for scaling up replicas and 30 seconds for scaling down. During this stabilization window, the autoscaling solutions cannot change the replica number.

In both tests, we code each worker-client as an infinite loop of requests. It sends a png image, encoded as UInt8Array, to the Tensorserving REST interface, exposed by Ingress-NGINX. We limit the rate at which workers can make requests to 2 requests per second to avoid high traffic density and degradation of requests. The semantic of clients requests is synchronous blocking, this means that each request waits for the previous one to be completed before sending. The tests last around 30 minutes each and the worker number starts from 1 and grows to a maximum of 8. The number of workers increases every 4 minutes. Before reaching two workers it is increased by one, by two after this threshold. This stress test aims to generate a high load of computation in the backend to let Polaris and HPA autoscale the system based on the given metrics.

D. 90th Percentile 500ms and 70% CPU Usage

Figure 1 shows the first test handled with 90th percentile of 500ms SLO for Polaris and 70% CPU Usage for HPA. The figure shows that the latency experienced by the considered application, handled by Polaris with high-level metrics (blue lines and left y-axis), present fewer spikes and generally lower values with respect to HPA (green line, right y-axis). When the number of workers increases and the requests per second become higher, we can see that Polaris scales out the pods respecting the SLO better than HPA. This is because

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on June 27,2022 at 06:55:32 UTC from IEEE Xplore. Restrictions apply.

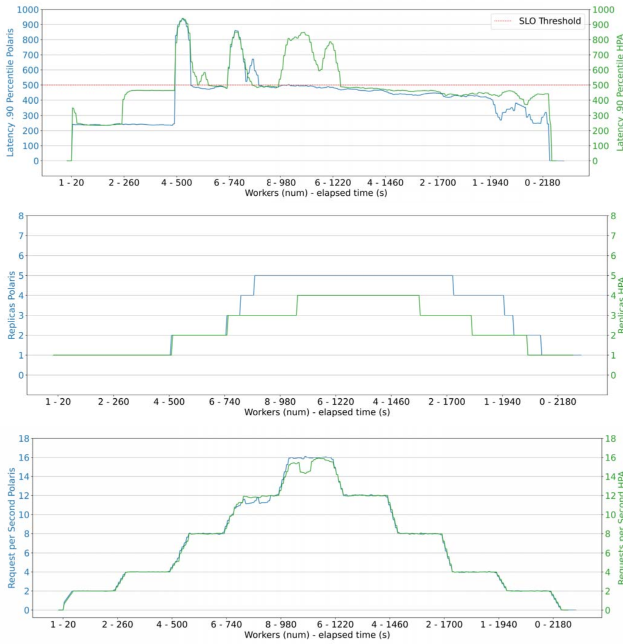

Fig. 1. Polaris 90th percentile 500ms latency SLO and HPA 70% CPU usage comparision

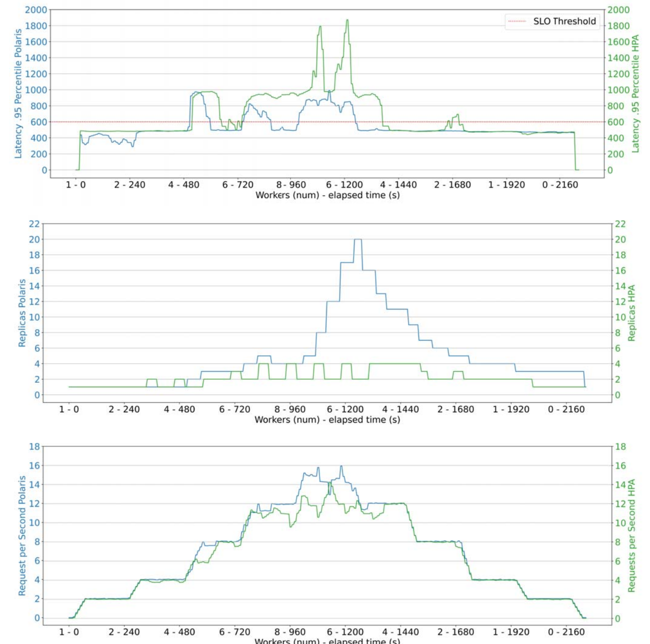

Fig. 2. Polaris 95th percentile 600ms latency SLO and HPA 50% CPU usage comparision

Polaris, by directly monitoring the latency SLO, knows better than HPA when the SLO is not fulfilled. Only to mention one practical example, around timestamp 720, Polaris holds additional pods in respect to HPA. Around 1920 elapsed time, when the switch from 2 to 1 worker occurs, we can also see that Polaris has been very reactive in pushing down the pods number once the SLO was over-respected, to save costs and resources.

E. 95th Percentile 600ms and 50% CPU Usage

Figure 2 shows the second set of tests that we have accomplished for Polaris and HPA. In these tests Polaris SLO have been set to 95th percentile of 600ms SLO and HPA to 50% CPU Usage.

Since a higher percentile is more volatile, we decided to increase the latency SLO to 600ms and to lower HPA to 50 percent of CPU Usage to let it scale more pods at once. In this figure, we can see that with 8 workers HPA is not able to understand that it has to scale out resources and, by not doing it, violates the SLO. On the other hand, Polaris is aware that our SLO is violated and increased the replica number up to 20 to improve the SLO Compliance value. Even if this requires additional resources, Polaris is still adhering to the SLO, which is the desired outcome in this case. Obviously, this is a trade-off that we must understand if we are to respect the SLO despite our extensive use of resources. For the purposes of this experimental evaluation, we focused on SLO compliance rather than resource utilization.

Similar to Figure 1, also Figure 2 shows how the high-level SLO scaling can be more reactive when the SLO is violated. For example, when the latency starts to go down, around time 1200, our SLO-based solution starts to scale down to save costs and free resources, thus confirming again our claim about the suitability of high-Level metrics for SLO-aware autoscaling.

VI. Conclusions

This work originally presents and evaluates the performance of Polaris as a high-level metric-based pods autoscaler for Kubernetes, by extensively comparing it to the commercial state-of-the-art HPA. In summary, this performance comparison shows that a high-level metric scaling can provide lower latency and more reactive decisions as the load changes. In terms of a functional comparison, Polaris has demonstrated to be a very complete autoscaler, it decouples Elasticity Strategies and SLO Controllers and, thus, enables the provider to change the SLO or the strategy to apply very fast and without redeploying the entire system. Polaris can be attached to any existing deployment environment orchestrated with Kubernetes without the need of changing the overall system configuration. It only needs to be deployed and configured by creating the needed controllers and mappings.

We identify several steps of future work to propose novel experimental results. Other autoscaling solutions, in addition to Polaris and HPA, can be studied and compared. Furthermore, a larger portfolio of high-level and low-level metrics can be compared. This could provide different perspectives of how these autoscaling systems behave as metrics change and what performance results they can lead to. Additionally, a larger

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on June 27,2022 at 06:55:32 UTC from IEEE Xplore. Restrictions apply.

capacity experimental testbed might be used to reproduce the experiments in more production-like environment.

References

[1] S. Dustdar, Y. Guo, B. Satzger, H.-L. Truong, ”Principles of elastic processes, Internet Computing”, IEEE 15 no. 5 (2011) 66–71.

[2] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, ”Container Orchestration Engines: A Thorough Functional and Performance Comparison”, IEEE International (2019).

[3] T. Nguyen, Y. Yeom, T. Kim, D. Park, S. Kim, ”Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration”, Sensors (Basel, Switzerland) 20 (2020) 1–18.

[4] J. Ding, R. Cao, I. Saravanan, N. Morris, C. Stewart, ”Characterizing service level objectives for cloud services: Realities and myths”, 2019 IEEE International Conference on Autonomic Computing (ICAC) (2019).

[5] A. Keller, H. Ludwig, ”The wsla framework: Specifying and monitoring service level agreements for web”, Journal of Network and Systems Management 11 (2003) 57–81.

[6] T. Pusztai, A. Morichetta, V. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, Y. Xiong, ”A Novel Middleware for Efficiently Implementing Complex Cloud-Native SLOs”, 2021 IEEE 14th International Conference on Cloud Computing (CLOUD) 21 (2021) 410–420.

[7] R. Buyya, C. S. Yeo, S. V. J. Broberg, I. Brandic, ”Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility”, Future Generation Computer Systems 24 no. 30 (2009) 599–616.

[8] E. F. Coutinhom, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, J. N. de Souza, ”Elasticity in cloud computing: a survey, annals of telecommunications annales des telecommunications” 70 no. 7-8 (2015) 289–309.

[9] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, Y. Xiong, ”SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs”, 2021 IEEE International Conference on Web Services (ICWS)

[10] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, ”Elasticity in Cloud Computing: State of the Art and Research Challenges”, IEEE Transactions on Services Computing, 11 n. 2, 2018.

[11] Amazon web services inc, ”AWS auto scaling features”, aws.amazon.com/autoscaling/features/ (2020).

[12] Microsoft, ”Autoscaling documentation”, docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling (2017).

[13] Google inc, ”Autoscaling groups of instances”, cloud.google.com/compute/docs/autoscaler (2020).

[14] Kubernetes, ”Vertical pod autoscaler”, github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler (2021).

[15] Kubernetes, ”Kubernetes autoscaler”, github.com/kubernetes/autoscaler (2021).

[16] TensorFlow, ”Mobilenetv2”, github.com/tensorflow/models/tree/master-/research/slim/nets/mobilenet (2017-2021)

[17] Amazon AWS, ”Amazon CloudWatch Percentiles on Amazon S3 brings more precision to tracking application response time metrics”, aws.amazon.com/it/blogs/storage/amazon-s3-cloudwatch-percentiles (2019)

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on June 27,2022 at 06:55:32 UTC from IEEE Xplore. Restrictions apply.