RAYGO: Reserve As You GO

Stefano Galantino, Marco Iorio, Fulvio Risso
Dept. of Computer and Control Engineering
Politecnico di Torino
Torino, Italy
{name.surname}@polito.it

Antonio Manzalini
Innovation Labs
Telecom Italia Mobile
Torino, Italy
antonio.manzalini@telecomitalia.it

***Abstract*—The capability to predict the precise resource requirements of a microservice-based application is a very important problem for cloud services. In fact, the allocation of abundant resources guarantees an excellent quality of experience (QoE) for the hosted services, but it can translate into unnecessary costs for the cloud customer due to the reserved (but unused) resources. On the other side, poor resource provisioning may turn out in scarce performance when experiencing an unexpected peak of demand. This paper proposes RAYGO, a novel approach for dynamic resource provisioning to microservices in Kubernetes that (i) reliefs the customers from the definition of appropriate execution boundaries, (ii) ensures the right amount of resources at any time, according to the past and the predicted usage, and (iii) operates at the application level, acknowledging the dependency between multiple correlated microservices.**

***Keywords*-Resource management, Cloud, container, autoscaling, vertical scaling**

I. Introduction

Many companies massively leverage cloud computing services for their IT businesses, possibly benefiting from the *pay-as-you-go* paradigm (also referred to as the serverless approach), using and paying only the resources needed at any given time. This can bring significant cost savings to customers and allow them to concentrate in their higher-level business logic, leaving infrastructure-level tasks (e.g., hardware provisioning and maintenance) to the cloud provider.

However, in order to share the infrastructure among the above customers, the cloud provider usually requires each tenant to define precise resource boundaries for their workloads. According to the Kubernetes terminology, this involves two parameters: *Requests* and *Limits*. A Request is what the microservice is guaranteed to get (e.g., 0.5 virtual CPUs), while the Limit determines the maximum amount of resources that can be consumed (e.g., 1 virtual CPUs), which can be provided in a best-effort fashion. In any case, the workload is never allowed to consume resources beyond limits. Finally, Requests and Limits could also be used by the cloud provider to charge the customer, either directly or through cluster autoscaling, which therefore has to carefully balance the trade-off between resource abundance and costs.

Still, developers may not know in advance how to quantify properly these execution boundaries, ending up in two common errors: (i) over-commitment, asking for an amount of resources that is way higher than the actual needs of the application, in order to guarantee the best QoE in every circumstance; (ii) under-commitment, defining boundaries lower than the actual needs, due to imprecise estimations or unexpected spikes of requests. In the first case (over-commitment), the customer will be charged also for unused resources, resulting in unnecessary expenses. In the second case (under-commitment), the customer will experience poor QoE for the deployed application. Interesting, the first scenario is the most likely: Google estimates an actual resource usage for application requests of no more than $50\,\%$ [1].

These issues trace back to the immutability of the resources assigned to an application throughout its execution, missing the possibility to update them at run-time. Yet, it is hard to assess in advance how many resources a job needs to run optimally. Load tests can help finding an initial estimate, but these recommendations become soon stale as workload needs change over time. Indeed, many end-user serving jobs have daily or weekly load patterns, and traffic changes across longer time scales as a service becomes more or less popular.

A common solution to the above problem is horizontal autoscaling, which is largely available in public cloud providers as well as in vanilla Kubernetes. This technique (e.g., Kubernetes Horizontal Pod Autoscaler — HPA [2]) leverages the dynamicity of the microservice paradigm by adding or removing replicas in response to changes in the metric under observation, such as the end-user traffic, the average CPU utilization, and more. However, despite its popularity, horizontal autoscaling may be difficult to configure properly and in many cases its effectiveness is subordinated to a significant waste of resources. In fact, the creation of new replicas implies a step increase of all reservations; furthermore this technique is sub-optimal in case only one type of resource (e.g. CPU) should be adjusted to address the current demands, as horizontal autoscaling implies adjusting resources of all types (e.g., CPU and RAM).

A less frequent approach involves vertical autoscaling (e.g., Kubernetes Vertical Pod Autoscaler — VPA [3]) to tune at run-time the amount of resources available to each replica by configuring the Requests of each container based on its

usage. Although apparently less popular, vertical autoscaling is adopted as a valuable solution for container management in Google data centers [1], providing a more accurate and fine grained control of resource provisioning. It ensures better performance especially for those applications that rely on specific network protocols for inter-microservice communication, such as gRPC, which suffer from poor load-balancing if the number of back-ends varies dynamically. In fact, gRPC massively relies on long-lived TCP persistent connections, which can hardly be split or redirected to other replicas, hence hindering the basic assumption under horizontal autoscaling. However, classical vertical autoscaling operates at the *single* microservice level, hence possibly neglecting the correlations between the multiple components of a single application. Additionally, it was designed with slowly-variable workloads in mind, hence failing to achieve good performance if the load changes abruptly.

The main contribution of this paper is RAYGO, a prediction algorithm for vertical resource provisioning in Kubernetes. RAYGO handles the execution of a microservice-based application through two components. First, a proactive engine, which estimates the future microservice resource demands based on its past history. Second, a reactive engine, which dynamically refines the profiling decision according to the overall behavior of the entire application, ensuring fast reactions and preventing performance drops during load spikes.

The rest of the paper is organized as follows. Section II details the RAYGO algorithm, as well as its experimental evaluation in Section III. Finally, Section IV summarizes the related work and Section V concludes the paper.

II. RAYGO

Given the complexity and the heterogeneity of microservices behavior, we designed RAYGO as composed of two different components. Specifically, (i) a *proactive* engine (Historical Data Profiler — HDP), which aims to predict the future behavior of a microservice based on its past executions and (ii) a *reactive* engine (Execution Data Predictor — EDP), which constantly monitors the entire application execution (i.e., the combination of multiple related microservices) and refines the profiling decision based on the information about its current behavior.

The combination of the above two components is required since a purely proactive approach may not be suitable for heavily variable workloads characterized by sudden spikes of requests and unexpected load changes. On the other hand, a purely reactive one, following strictly the current needs of the jobs, may be able to identify load changes much quicker, but it can result in continuous updates of the resources assigned to each microservice. Yet, as of today and in the context of Kubernetes clusters, this is a highly disruptive process, given it requires to restart the application. Although compliant with the stateless approach, too frequent updates could result in poor QoE, as well as overall higher resource consumption during the transient. Conversely, a purely reactive approach might be feasible if the resources assigned could be varied at run-time, with no service disruption. Overall, the role of the proactive component is to mitigate the update rate of the reactive one, hence limiting the application downtime.

A. Historical Data Profiler (HDP)

One of the results emerging from major cloud provider reports [4] is that many hosted applications experience quite stable resource usage patterns in relatively small periods of time. Hence, we can expect the future behavior of a given microservice to be similar to the one observed in the past. Specifically, this component (i) collects the information about the previous executions of a given microservice, (ii) processes the historical data to extract the relevant key features and (iii) leverages the outcome to compute the final profiling value.

Focusing on the feature extraction phase, let consider as input, at a specific instant in time t_0, the set of the n past measurements $\xi_{\mu,x}$ referred to a given resource quantity x (e.g. CPU usage) of microservice μ and collected every δ s:

$$\Xi_{\mu,x}(n) = \{\xi_{\mu,x}[t_0 - \delta i] : i \in [0, n)\}. \tag{1}$$

First, the measurements are weighted by the function $w[i] = 2^{-\delta i/\tau}$, to smooth the response to load spikes and give increased relevance to the samples closer in time:

$$\hat{\Xi}_{\mu,x}(n) = \Xi_{\mu,x}(n) \cdot w[i] = \{\xi_{\mu,x}[t_0 - \delta i] \cdot 2^{-\frac{\delta i}{\tau}} : i \in [0, n)\}, \tag{2}$$

where τ is defined as the half life, that is the time after which the weight drops by half. In other words, the larger τ, the slower the system reacts, due to the increased relevance associated with the older samples. Conversely, a small τ value corresponds to more rapid reactions, quickly neglecting past measurements.

At this point, the final HDP prediction $\Phi^{\text{HDP}}_{\mu,x}$ is computed as the r^{th} percentile (P_r) of the last n weighted samples:

$$\Phi^{\text{HDP}}_{\mu,x} = \underset{i}{\text{P}_r}(\hat{\Xi}_{\mu,x}(n)), \quad i \in [0, n). \tag{3}$$

The selection of the appropriate r value needs to necessarily take into account the specific characteristics of the monitored feature (e.g., its volatility), as well as the orchestrator behavior. For instance, an under-sized CPU limit could simply lead to reduced performance, as an orchestrator can enforce throttling periods to satisfy the CPU limits. This does not raise any concern for sufficiently short intervals as computations can complete correctly, just slightly slower. Instead, an under-sized RAM limit could lead to service disruption, as a microservice exceeding its limits causes the orchestrator to react with an out-of-memory (OOM) event, effectively terminating the microservice and causing the restart of the application. The following details the approach selected for CPU and RAM predictions, although similar considerations

apply in case other resource types (e.g. ephemeral storage) are considered.

CPU usage: Given the possible high volatility of this metric due to the alternations between short-term load spikes and idle periods, it is fundamental to balance between QoE and excessive resource demands triggered by load peaks. For this reason, based on our experience, we suggest to select $r \in [90, 100]$, with lower values resulting in more aggressive resource estimations, and higher ones being more conservative, reducing the probability of throttling periods:

$$\Phi^{\text{HDP}}_{\mu,\text{CPU}} = \mathrm{P}_{i}{}_{r}(\hat{\Xi}_{\mu,x}(n)), \quad i \in [0, n), r \in [90, 100]. \tag{4}$$

RAM usage: Given the service disruption possibly experimented with a poor RAM estimation and considering that the RAM usage typically varies slowly over time due to allocation and caching policies, we conservatively suggest to select $r = 100$. Hence, the HDP module computes $\Phi^{\text{HDP}}_{\mu,\text{RAM}}$ as the *maximum* of the last n weighted samples:

$$\Phi^{\text{HDP}}_{\mu,\text{RAM}} = \max_{i} \hat{\Xi}_{\mu,x}(n), \quad i \in [0, n). \tag{5}$$

B. Execution Data Predictor (EDP)

The second concept which emerges from public reports [4] regards the impossibility to correctly infer the future resource consumption based on historical data only for certain categories of microservices. In these cases, it is clear that the proactive approach adopted by the HDP, alone, is not sufficient.

For this reason, we introduce the Execution Data Predictor (EDP) engine, which continuously adapts the HDP prediction based on the current application demands. Specifically, it (i) constantly monitors a different set of metrics (as detailed in the following) with respect to the HDP, to characterize the current microservice execution; (ii) uses the gathered information to identify load spikes; (iii) generates a corrective factor to adapt $\Phi^{\text{HDP}}_{\mu,x}$ based on the specific resource demands. The EDP operates considering the *complete* set of microservices composing a single application, rather than assessing each one *independently*. Indeed, preliminary evaluations have shown a reduction in the number of application resource updates with such approach and consequently significant improvements in application performance (thanks to the downtime reduction). Indeed, without the contextualized view of the application RAYGO struggles to identify resource predictions, alternately updating the microservices as the workload pressure moves constantly from the front-end to the back-end, and viceversa.

As for the metrics considered, the EDP focuses specifically on the indicators highlighting that a given set of microservices is struggling to achieve good performance. In detail, as for CPU usage, we consider the amount of throttling periods imposed by the orchestrator, which points out the presence of too strict limits with respect to the current workload demands. Similarly, considering RAM usage, the EDP evaluates the number (and type) of memory failure events (ranging from page faults to OOM) sent to the microservice, as we experimentally observed they tend to grow when reaching the configured boundaries.

Let $\Xi'_{\mu,x}(m)$ represent again the set of the past m measurements, considering in this case the number of throttling periods (CPU prediction) and memory failure events (RAM prediction) as metrics of interest ($\xi'_{\mu,x}$). Considering sufficiently short time intervals, we can assume these indicators to follow a linear trend. Hence, we adopt a linear regression model to predict the evolution of the values. Leveraging the least square method, the predicted value $\xi'_{\mu,x}[t_p]$, where $t_p = t_0 + p\delta$ and $p > 0$, can be estimated from the previous m observations as:

$$\xi'_{\mu,x}[t_p] = 2\frac{\alpha \sum_{i=0}^{m-1} \xi'_{\mu,x}[t_0 - \delta i] - \beta \sum_{i=0}^{m-1} i\xi'_{\mu,x}[t_0 - \delta i]}{m(m^2-1)}, \tag{6}$$

where:

$$\alpha = 2m^2 - 3m + 3mp - 3p + 1 \quad \text{and} \quad \beta = 3(m + 2p - 1). \tag{7}$$

Next, for each metric considered, we derive the application-wide baseline value $\Upsilon_{A,x}$, to capture the behavior of the entire set of related microservices $\mu \in A$. Specifically, $\Upsilon_{A,x}$ is computed as the average of the last m measurements, over all microservices composing the application of interest:

$$\Upsilon_{A,x} = \operatorname*{avg}_{\mu,i} \Xi'_{\mu,x}(m), \quad i \in [0, m), \mu \in A. \tag{8}$$

Given the outcome of the prediction $\xi'_{\mu,x}[t_p]$ and the baseline $\Upsilon_{A,x}$, we derive, for each microservice μ, an intermediate factor $\Psi_{\mu,x}$ obtained computing the ratio between the two resulting values, in a way that the result is always a number ≥ 1.

$$\Psi_{\mu,x} = \begin{cases} \frac{\xi'_{\mu,x}[t_p]}{\Upsilon_{A,x}}, & \text{if } \xi'_{\mu,x}[t_p] \geq \Upsilon_{A,x} \\ \frac{\Upsilon_{A,x}}{\xi'_{\mu,x}[t_p]}, & \text{if } \xi'_{\mu,x}[t_p] < \Upsilon_{A,x} \end{cases} \tag{9}$$

In the end, the outcome of the EDP engine ($\Phi^{\text{EDP}}_{\mu,x}$) is:

$$\Phi^{\text{EDP}}_{\mu,x} = \begin{cases} +\lambda e^{\sigma|\Psi_{\mu,x}|}, & \text{if } \xi'_{\mu,x}[t_p] \geq \Upsilon_{A,x} \\ -\lambda e^{\sigma|\Psi_{\mu,x}|}, & \text{if } \xi'_{\mu,x}[t_p] < \Upsilon_{A,x} \end{cases} \tag{10}$$

where λ and σ are two positive scaling constants. The sign of $\Phi^{\text{EDP}}_{\mu,x}$ reflects the predicted additional demands of the current microservice compared to the entire application. Intuitively, focusing on CPU usage, above-average throttling periods indicate a struggling microservice (i.e., demanding for more resources), while below-average ones typically follow the end of a load spike, thus allowing for stricter quotas.

C. Final Resource Prediction

Given $\Phi^{\text{HDP}}_{\mu,x}$ and $\Phi^{\text{EDP}}_{\mu,x}$, the final resource prediction $\Phi_{\mu,x}$ for the microservice μ and resource quantity x is derived as:

$$\Phi_{\mu,x} = \Phi^{\text{HDP}}_{\mu,x} \cdot (1 + \Phi^{\text{EDP}}_{\mu,x}). \tag{11}$$

Table I
THE RAYGO PARAMETERS CONFIGURATION.

Prediction evaluation period (Δ):	$2\,\text{min}$
Measurements sampling period (δ):	$1\,\text{s}$
Resource Requests increase factor (ρ):	0.1
Resource Limits increase factor (ν):	0.7
HDP measurements rolling window size (n):	900
HDP measurements rolling window duration ($n\delta$):	$15\,\text{min}$
HDP measurements half life (τ):	$15\,\text{min}$
HDP CPU prediction percentile (r):	97^{th}
HDP RAM prediction percentile (r):	100^{th}
EDP measurements rolling window size (m):	120
EDP measurements rolling window duration ($m\delta$):	$2\,\text{min}$
EDP forward prediction samples (p):	15
EDP scaling constant (σ):	0.5
EDP scaling constant (λ):	0.1

Overall, the final result is composed of two parts: a baseline, represented by the $\Phi^{\text{HDP}}_{\mu,x}$ value, and a corrective factor (either positive or negative) predicted by the EDP engine.

The entire process is repeated every $\Delta\,\text{s}$, hence periodically recomputing new $\Phi_{\mu,x}$ predictions values based on the updated information and possibly varying the microservices configuration depending on the outcome. Hence, a known microservice is started with the latest $\Phi_{\mu,x}$ values, while its resource quota is periodically updated according to the most recent (application-wide) predictions in order to match the actual necessities of the application.

III. EXPERIMENTAL VALIDATION

This section validates the above approach through a prototype implementation of RAYGO, publicly available at [5]. It manages the microservices execution within a Kubernetes cluster by dynamically adjusting the amount of resources assigned to each single workload according to the outcome of (11). Specifically, *Requests* are configured to the $\Phi_{\mu,x}$ value, possibly incremented by a small, user-configurable safety margin $\rho \geq 0$ (i.e., $R_{\mu,x} = \Phi_{\mu,x} \cdot (1+\rho)$). Then, *Limits* are obtained enlarging $R_{\mu,x}$ by a configurable factor $\nu \geq 0$ (i.e., $L_{\mu,x} = R_{\mu,x} \cdot (1+\nu)$), to account for sudden and temporary load spikes without waiting for the reaction of RAYGO, which may be slower (and possibly unnecessary). TABLE I details the values of the complete RAYGO parameters adopted for the evaluation.

In Kubernetes, one of the most adopted abstractions to execute microservices is the *Deployment*. Deployments define the template of the microservice, including the Docker image, its associated execution environment and, most importantly in this context, the amount of resources (in terms of Requests and Limits) that are enforced by the orchestrator. Additionally, they ensure the desired number of replicas is correctly in execution, and transparently manage rolling updates (i.e., starting a new parallel instance of the microservice and tearing down the old one only once the former is correctly running) to limit the application downtime in case the template is varied.

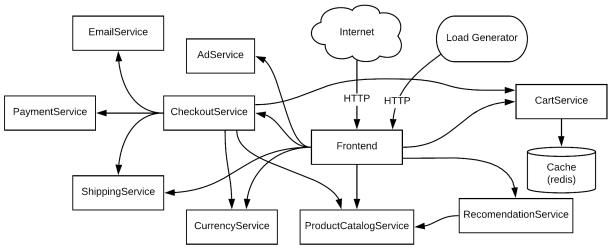


Figure 1. A representation of the microservices composing the Online Boutique, and their interconnections.

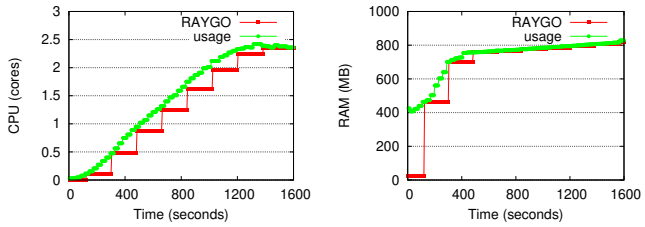
Our implementation leverages standard Kubernetes labels to identify the set of microservices composing a single application, and relies on Prometheus[1] to gather the execution metrics used by the algorithm. This prototype could be easily extended to properly manage other Kubernetes abstractions (e.g. StatefulSets), as well as to directly gather the metrics from the *metric-server*[2] if Prometheus was not available.

A. Testbed setup

In the validation process, we leveraged the cloud-native demo application Online Boutique[3], which consists of ten microservices (graphically represented in Fig. 1) interacting among them through gRPC interfaces. The application is a web-based e-commerce allowing users to browse and purchase items, while featuring recommendations and currency exchange. Locust[4], an open-source load testing tool, was used to replicate end-user interactions with the platform. It allows to define a custom application workload, in terms of number of fake clients and target endpoints, and then generates the suitable requests according to the configuration.

For the sake of comparison, we first evaluated the performance of the application running unsupervised, without restrictions in resource usage. Then, we assessed the outcome when managed by the Vertical Pod Autoscaler (`VPA`), v0.9.2, as well as the Horizontal Pod Autoscaler (`HPA`), as of Kubernetes 1.19, configured to increase the number of replicas when the CPU load reaches $80\,\%$. Finally, we tested RAYGO, considering the case with only HDP and the combination of the two engines. In the latter case, we evaluated both a degraded version of the EDP module, assessing each microservice independently (HDP+EDP single), as well as the full, application-aware one (HDP+EDP app). Across all tests, the amount of resources available on the nodes was more than enough to accommodate all the microservices, hence posing no constraints on the different approaches.

[1] https://prometheus.io/
[2] https://github.com/kubernetes-sigs/metrics-server
[3] https://github.com/GoogleCloudPlatform/microservices-demo
[4] https://locust.io/



(a) (b)

Figure 2. Actual CPU and RAM usage, along with values tracked by RAYGO, during the initial warm-up phase.

B. Workload Pattern

Using Locust, we generated a custom workload characterized by two main phases. In a first *warm-up phase*, the number of simulated users is gradually increased up to 1000, and then kept constant. This allows all solutions to adapt the amount of resources (and possibly replicas) based on the application demands. At that point, the actual *test phase* starts, which simulates an unforeseen spike of users to assess how the different application supervisioning approaches behave in this scenario. Specifically, it is characterized by four intervals: first, the number of simulated users grows linearly (one new user every second) up to 2000 users, and then is maintained constant. In the third phase, the users start decreasing (one user removed every second) until reaching again 1000, followed once more by a stationary phase. Hence, assessing both the behaviour, in terms of QoE and resources, during and after a load spike.

Fig. 2 shows the adaptation process performed by RAYGO during the warm-up phase, regarding CPU and RAM usage (represented as the total across all microservices), with the predictions closely tracking the actual values while the number of users (and hence the load) increases. Being the given application CPU intensive, the following evaluation focuses specifically on this metric. However, similar results have been obtained regarding memory usage.

C. Resource usage

Figs. 3 and 4 show respectively the sum of the CPU requests and limits assigned to the set of microservices by the different approaches. Results are aggregated by test phase, with the bar graph showing the average, and the vertical segment representing the minimum and maximum values. Overall, RAYGO achieved significant resource savings compared to the other approaches. Indeed, the HPA suffered from the suboptimal step increases caused by the creation of new replicas, while the VPA struggled to precisely track the application load, keeping the resource demands high even after the spike of requests terminated.

In this context, the additional usage of the application-aware EDP engine, thanks to its ability to detect unexpected load changes, resulted in more resources assigned to the microservices during the workload growing phase, and in slight resource savings when the number of generated users decreased, ensuring better performance for the application, as detailed in the following.

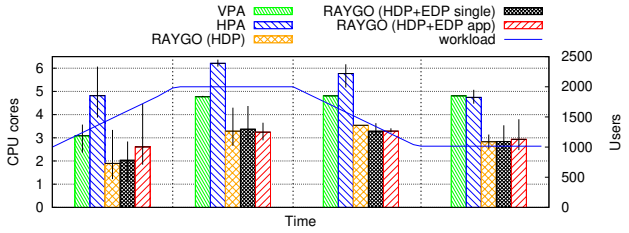


Figure 3. The sum of CPU requests assigned by each solution to the set of microservices, grouped by test phase.

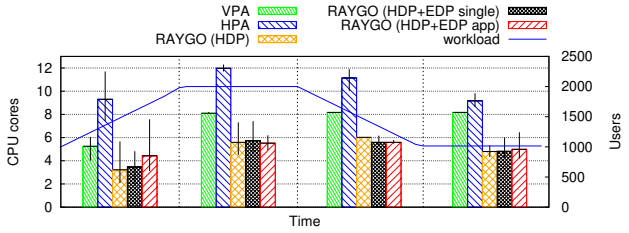


Figure 4. The sum of CPU limits assigned by each solution to the set of microservices, grouped by test phase.

D. Quality of Experience

Previous results could raise the question whether the smallest limits requested by RAYGO are enough to sustain the application workload. In this section we evaluate the QoE provided by the application when supervised by the different solutions. First, we assessed the number of requests correctly processed (i.e., with a successful `2XX` response), and the outcome is presented in Fig. 5. The *unconstrained* values represent a reference measured with no resource constraints, hence reflecting the maximum achievable in the specific test conditions. Given the behavior of Locust, with each client issuing a new request only after the previous response is received (or a timeout expired), a lower result can derive from both errors (e.g. a microservice is being restarted), as well as increased application response times.

During the entire evaluation, RAYGO executed with the combination of HDP and EDP resulted the solution ensuring the performance closest to the reference, despite the overall lower resource demands (cf. Fig. 3). Furthermore the contextualized decisions of the EDP, working with the complete set of microservices, has shown significant improvements in the QoE. Indeed, during the initial phases, the VPA, RAYGO (HDP only) and RAYGO (HDP+EDP single) struggled to adapt the configuration to the workload demands, recovering only when the number of users decreased. While requiring the highest amount of resources, the application supervised by the HPA displayed a $5\,\% - 10\,\%$ gap behind the reference. Although unexpected, this result is caused by the usage of the gRPC protocol for the interaction between microservices.

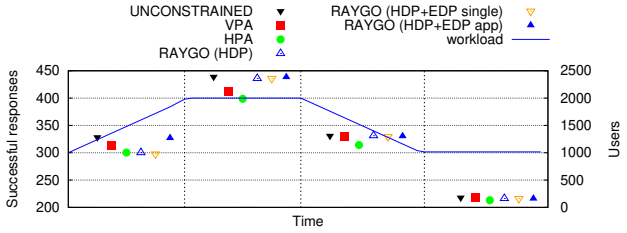


Figure 5. Number of responses correctly served by the application either unsupervised or managed by the different approaches.

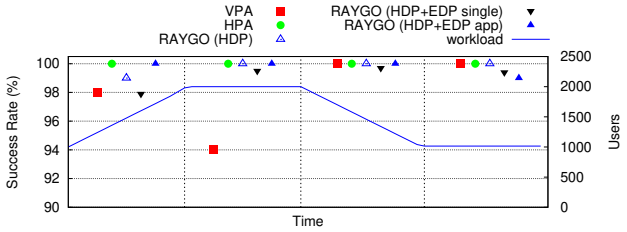


Figure 6. Responses success rate.

Indeed, leveraging persistent connections, it fails to properly load balance the requests when new back-end replicas are created at run-time by the HPA. This problem could be addressed by service mesh techniques, though at the cost of increased complexity and resource consumption.

Additionally, Fig. 6 depicts the success rate of the application managed by the three different solutions, that is the number of 2XX responses out of the total received ones. Errors may be returned both in case one of the microservices was temporarily unavailable (e.g., while being restarted) or the request exceeded the 5 s timeout, hence preventing the completion of the transaction. In a nutshell, the vertical autoscaling-based approaches suffered the most, due to the microservice restarts required to adapt the resources. Still, RAYGO strongly limited the disruption, thanks to its application-aware approach. Conversely, much lower success rate was displayed by the VPA, due to uncoordinated updates.

To assess the overall trade-off between QoE and resource demands, Fig. 7 presents the ratio between the number of successful responses achieved by each solution and the resource requests configured (although the outcome is consistent if limits are considered). Overall, RAYGO stands out as the algorithm characterized by the best results during each phase of the test. Yet, the results of RAYGO (HDP only) and RAYGO (HDP+EDP single) are partially misleading, especially in the first test phase, where they show the highest efficiency. However, the limited amount of assigned resources (Fig. 3) comes at the expense of a poor QoE, as shown by the bad result in terms of successful responses (Fig. 5).

E. Response time

Finally, we analyzed the responsiveness of the application when responding to user requests. Fig. 8 presents through a box-plot the distribution of the 95th percentile (P_{95}) of

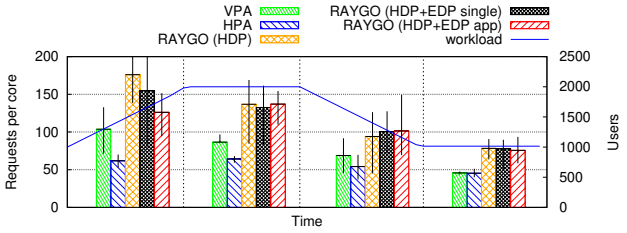


Figure 7. Ratio between the number of successful application responses and the resource requests configured by each solution.

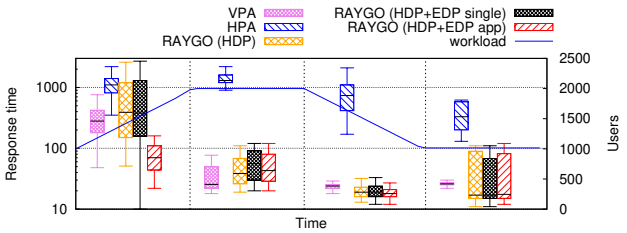


Figure 8. Distribution of the P_{95} of the response times, considering one-second long intervals, over time.

the response times, considering one-second long intervals. In other words, every 1 s the P_{95} value is extracted from all responses received, and the resulting samples, grouped by test phase, constitute the box-plot. In this case, the HPA is the solution suffering the most, once more due to the inefficient load-balancing of gRPC traffic. Indeed, new requests continued to be issued to the overloaded replicas, although new available instances had been created by the HPA. Differently, both VPA and RAYGO (HDP+EDP app) were able to guarantee significantly faster response times (i.e. $\ll$ 1 s for RAYGO) with the median value of the P_{95} distribution never exceeding 100 ms in case of RAYGO. Furthermore, the application-aware EDP engine showed its effectiveness in reducing the overall response time, especially during load spikes.

IV. Related work

Many papers and reports already analyzed the most common workloads in public data centers, aiming at identifying recurrent patterns in resource usage [4], [6], [7] and enabling researchers to explore how scheduling works in large-scale production compute clusters on a long time scale. In this scenario the problem of resource autoscaling for microservices is definitely one of the most relevant topics: indeed, the most adopted solutions for container orchestration (i.e., Kubernetes, Borg [8] and YARN [9]) require users to specify the amount of resources assigned to each job. Once properly defined these values, despite the multi tenancy property of the cloud environment and the underlying shared infrastructure, the orchestrator can guarantee to each application its reserved slice of resources.

Besides widespread open-source solutions such as the Kubernetes HPA and VPA already mentioned in Section I,

autoscaling is a well-developed research area. Many papers addressed the problem of horizontal scaling of resources: [10] introduces a modularized platform for resource provisioning, while [11] defines probabilistic performance models of horizontal autoscalers both in AWS and Azure and [12] exploits an absolute CPU utilization correlation model to accurately predict the number of replicas. Moving to vertical autoscaling, a key enabler is represented by resource usage prediction models. To this end, different approaches have been proposed, including time-series forecasting based on a second order autoregressive moving average method (ARMA) [13], the computation of the median of resource usage observations [14], neural networks [15] and reinforcement learning techniques [16]. Yet, in many scenarios, the computational overhead, as well as the additional time required in training the reinforcement learning network and the extreme variability of microservices behaviour, make those solutions not completely appropriate.

V. Conclusions and future work

The automatic prediction of the resource requirements of microservice-based applications is of fundamental importance to achieve the best trade-off between the amount of resources reserved (and hence charged) for their execution and the offered QoE. RAYGO is an algorithm tracking the resource demands of a set of applications and dynamically adapting their configuration according to the foreseen requirements. It combines a proactive approach, predicting future resource demands based on past executions, and a reactive component, which continuously refines the profiling decisions based on the current behavior of the entire application. We compared a RAYGO prototype with two common autoscaling mechanisms leveraged in Kubernetes, namely HPA and VPA, for the management of a realistic ten-tier microservices application. Overall, the results are promising, with RAYGO achieving both stricter resource boundaries and better QoE, in terms of successful responses and response time.

As future work, RAYGO could consider bandwidth demands in addition to CPU and RAM, hence recognising the communication patterns between closely-related microservices. Additionally, we can explore its integration with a job scheduler optimised for distributed edge scenarios.

Acknowledgment

The authors warmly thank prof. Guido Marchetto for his precious help in modelling the system described in this paper.

References

[1] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. on Computer Systems*, Apr. 2020, pp. 1–16.

[2] G. Saenger and G. Rodrigues, "Kubernetes horizontal pod autoscaler (HPA): design proposal," https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/horizontal-pod-autoscaler.md.

[3] K. Grygiel and M. Wielgus, "Kubernetes vertical pod autoscaler (VPA): design proposal," https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md.

[4] M. Tirmazi *et al.*, "Borg: The next generation," in *Proc. 15th European Conf. on Computer Systems*, Apr. 2020, pp. 1–14.

[5] S. Galantino, "RAYGO: Reserve as you go," https://github.com/SteGala/RAYGO.

[6] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on Alibaba cluster trace," in *Proc. 2017 IEEE Int. Conf. on Big Data*, Dec. 2017, pp. 2884–2892.

[7] C. Curino *et al.*, "Hydra: a federated resource manager for data-center scale analytics," in *Proc. 16th USENIX Symp. on Networked Systems Design and Implementation (NSDI 19)*, Feb. 2019, pp. 177–192.

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *ACM Queue*, vol. 14, no. 1, p. 70–93, Jan. 2016.

[9] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. on Cloud Computing*, Oct. 2013, p. 1–16.

[10] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2017, pp. 1–6.

[11] A. Evangelidis, D. Parker, and R. Bahsoon, "Performance modelling and verification of cloud-based auto-scaling policies," in *Proc. 17th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 355–364.

[12] E. Casalicchio, "A study on performance measures for auto-scaling cpu-intensive containerized applications," *Cluster Computing*, vol. 22, no. 3, pp. 995–1006, Jan. 2019.

[13] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. IEEE 4th Int. Conf. on Cloud Computing*, Jul. 2011, pp. 500–507.

[14] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *Proc. IEEE 12th Inter. Conf. on Cloud Computing*, Jul. 2019, pp. 33–40.

[15] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, Jan. 2012.

[16] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. on Cloud Computing*, Jul. 2019, pp. 329–338.