SLO and Cost-Driven Container Autoscaling on Kubernetes Clusters

Angelo Marchese[a] and Orazio Tomarchio[b]
Dept. of Electrical Electronic and Computer Engineering, University of Catania, Catania, Italy

Keywords: Cloud Computing, Container Technology, Kubernetes Autoscaler, Service Level Objectives, Cost Monitoring.

Abstract: Modern web services must meet critical non-functional requirements such as availability, responsiveness, scalability, and reliability, which are formalized through Service Level Agreements (SLAs). These agreements specify Service Level Objectives (SLOs), which define performance targets like uptime, latency, and throughput, essential for ensuring consistent service quality. Failure to meet SLOs can result in penalties and reputational damage. Service providers also face the challenge of avoiding over-provisioning resources, as this leads to unnecessary costs and inefficient resource use. To address this, autoscaling mechanisms dynamically adjust the number of service replicas to match user demand. However, traditional autoscaling solutions typically rely on low-level metrics (e.g., CPU or memory usage), making it difficult for providers to optimize both SLOs and infrastructure costs. This paper proposes an enhanced autoscaling methodology for containerized workloads in Kubernetes clusters, integrating SLOs with a cost-driven autoscaling policy. This approach overcomes the limitations of conventional autoscaling by making more efficient decisions that balance service-level requirements with operational costs, offering a comprehensive solution for managing containerized applications and their infrastructure in Kubernetes environments. The results, obtained by evaluating a prototype of our system in a testbed environment, show significant advantages over the vanilla Kubernetes Horizontal Pod Autoscaler.

1 INTRODUCTION

Microservices architecture is a widely used architectural style for enterprise software that breaks down large applications into a series of small, modular, independently deployable microservices (Salii et al., 2023).

For such applications, the distribution of workload across a cluster of servers can be achieved in a horizontal manner, obviating the necessity for a single, costly server. The allocation of resources can be managed with precision by replicating or allocating greater resources to microservices experiencing the highest demand or those requiring greater reliability.

It is evident that microservice applications exhibit the characteristics that render them as ”cloud-native”, that is to say, they possess the capability to execute and expand within contemporary, evolving environments such as public, private and hybrid clouds (Hongyu and Anming, 2023).

Services developed and organized in such a way, are complex systems designed to meet a wide range of non-functional requirements that are critical to their business operations, including service availability, responsiveness, scalability, and reliability. These requirements are typically formalized through Service Level Agreements (SLAs) between service providers and consumers, which outline the operational boundaries within which a service must perform. An SLA consists of one or more Service Level Objectives (SLOs), which define high-level performance indicators that must be maintained throughout the service delivery period. These indicators represent the desired state of service, such as uptime, latency, or throughput, and are essential for ensuring consistent service quality. Failing to meet SLOs can result in contractual penalties and damage to the service provider’s reputation.

Furthermore, it is also important, from 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 (Gupta et al., 2017). Service providers are then challenged to find the right balance between meeting SLOs and optimizing resource usage and costs.

The adoption of cloud computing technology

[a] https://orcid.org/0000-0003-2114-3839
[b] https://orcid.org/0000-0003-4653-0480

Marchese, A. and Tomarchio, O.
SLO and Cost-Driven Container Autoscaling on Kubernetes Clusters.
DOI: 10.5220/0013482100003950
In *Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025)*, pages 72-79
ISBN: 978-989-758-747-4; ISSN: 2184-5042

and service orchestration systems has emerged as a promising solution to address this challenge, thanks to the cloud infrastructure's reliability, availability, scalability, and elasticity, as well as the automation capabilities in service management provided by orchestration systems (Mukherjee et al., 2024; Calcaterra et al., 2021). In particular, the service autoscaling mechanisms offered by orchestration systems, along with the ability to rapidly provision and de-provision cloud infrastructure, are key features for finding the right balance between meeting SLOs and optimizing costs.

Service autoscaling mechanisms help minimize over or under provisioning issues by dynamically adjusting the number of service replicas to match the current user request load (Chen et al., 2018). However, most orchestration solutions only allow to scale the service based on the value of a low-level metric (i.e., a metric considering low-level monitoring indicators such as service CPU or memory usage) and in this way it is hard for a service provider to control the required high-level SLO and to optimize the infrastructure costs.

In this paper we propose an enhanced autoscaling methodology, with particular reference to containerized workloads utilizing Kubernetes clusters. The proposed approach addresses the limitations of conventional autoscaling methodologies by integrating an SLO and a cost-driven autoscaling policy. This facilitates more efficient autoscaling decisions that balance both service-level requirements and operational costs, offering a more comprehensive solution for managing containerized applications in Kubernetes environments and the infrastructure required for their execution.

The rest of the paper is organized as follows. Section 2 provides some background information about the Kubernetes autoscaling policy and discusses in more detail some of its limitations that motivate our work. Section 3 presents our proposed approach, detailing its implementation, while Section 4 discusses the evaluation results from a testbed environment. Related works are reviewed in Section 5, and Section 6 concludes the work.

2 KUBERNETES AUTOSCALING

Kubernetes is today the de-facto orchestration platform for the lifecycle management of containerized applications deployed on large-scale node clusters (Kubernetes, 2024; Gannon et al., 2017). A typical Kubernetes cluster comprises a control plane and a set of worker nodes. The control plane encompasses various management services running within one or more master nodes, while the worker nodes serve as the execution environment for containerized workloads. In Kubernetes, the fundamental deployment units are Pods, each containing one or more containers and managed by a Deployment resource.

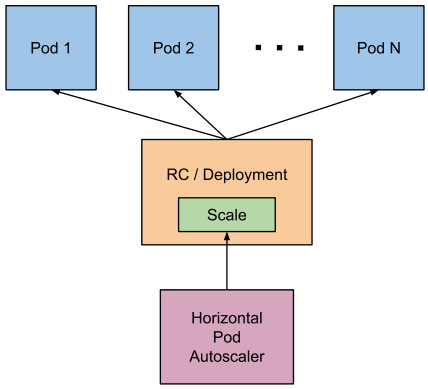

Figure 1: Kubernetes Horizontal Pod Autoscaler.

The Kubernetes Horizontal Pod Autoscaler (HPA) is a control plane component that adjusts the number of Pods managed by a Deployment based on the average CPU or memory usage of those Pods. Configured with a target value, the HPA periodically scales the number of Pods to ensure that their average resource usage aligns with the desired target. Equation 1 shows the HPA algorithm with a target value on the average CPU usage.

$$r_d = ceil(r_c * (cpu_c / cpu_d)) \quad (1)$$

where r_d is the desired number of replicas, r_c the current number of replicas, cpu_c the current average CPU usage, cpu_d the desired CPU usage and $ceil()$ is a function that gives as output the greatest integer less than or equal to the input argument. The main limitation of the HPA algorithm is that it relies on target values specified for low-level resource usage metrics, which are challenging to define and correlate with high-level indicators such as SLOs and infrastructure costs. Incorrectly defining these target values can lead to inefficient scaling decisions. Setting low resource usage targets triggers frequent scaling up actions, improving service performance but resulting in infrastructure over provisioning and higher costs. On the other hand, setting high targets results in frequent scaling down actions, reducing costs but potentially causing frequent SLOs violations.

3 PROPOSED APPROACH

3.1 General Model

Building on the limitations discussed in Section 2, this work introduces an SLO and cost-driven autoscaling policy specifically designed for containerized workloads running on Kubernetes clusters. The primary goal is to address the shortcomings of traditional autoscaling methods, such as the Kubernetes Horizontal Pod Autoscaler (HPA), which typically relies on low-level service resource usage metrics (e.g., CPU and memory utilization). These methods require continuous monitoring and configuration with target values for service resource consumption, and scaling decisions are based on achieving these predefined targets.

Our proposed autoscaling policy, whose general model is shown in Figure 2, aims to move beyond this resource-centric approach by integrating performance objectives (SLOs) and cost considerations into the scaling process. This enables more efficient autoscaling decisions that balance both service-level requirements and operational costs, offering a more comprehensive solution for managing containerized applications in Kubernetes environments and the infrastructure required for their execution.

The core idea of the proposed approach is based on the principle that an effective service autoscaling policy must ensure acceptable performance while minimizing infrastructure costs, especially in the face of fluctuating user request workloads. To achieve this, the policy should continuously monitor both the service's response time and the associated infrastructure costs through a monitoring framework, adjusting the system to meet predefined targets for these metrics. By dynamically balancing performance and cost, the approach aims to optimize resource allocation in real-time, ensuring both high-quality service delivery and cost-efficiency. Further details on the proposed custom autoscaler and the monitoring framework are provided in the following subsections.

3.2 Custom Pod Autoscaler

The proposed custom Pod autoscaler operates as a Deployment within the Kubernetes control plane and is built on top of the open source Custom Pod Autoscaler framework[1].

The autoscaler is configured through a *CustomPodAutoscaler* Kubernetes custom resource which define a scaling configuration for a Deployment. A CustomPodAutoscaler resource, whose schema is shown in Listing 1, contains a *spec* property with five sub-properties: *scaleTargetRef*, *runPeriod*, *stabilizationWindow*, *cost* and *slo*. The scaleTargetRef property identifies the target Deployment to scale. The runPeriod property determines the time interval, in milliseconds, between two consecutive executions of the autoscaling algorithm. The stabilizationWindow property defines the time interval, in milliseconds, following a scaling action during which the autoscaler cannot take further scaling actions for the Deployment. The cost property represents the desired target for the overall hourly cost of the cluster nodes. The slo property specifies a target SLO for the response time of the Deployment and contains three sub-fields: *p*, *window* and *target*. The target field represents the target SLO value, in milliseconds, for the *p*-quantile of the Deployment response time over the time period defined by the *window* field.

```yaml
apiVersion: custompodautoscaler.com/v1
kind: CustomPodAutoscaler
metadata:
  name: nginx-autoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  runPeriod: 30000
  stabilizationWindow: 10000
  cost: 100
  slo:
   p: 99th
   window: 20
   target: 300
```

Listing 1: Example of a CustomPodAutoscaler resource.

For each periodic execution of the autoscaling algorithm the number of replicas for the Deployment is determined by Equation 2.

$$r_d = ceil(k * r_c) \tag{2}$$

with:

$$k = w_{rt} * (rt_c/rt_d) + w_c * (c_d/c_c) \tag{3}$$

where r_d is the desired number of replicas, r_c the current number of replicas and k a multiplier factor. The value of the k parameter is determined as the weighted average between the response time SLO ratio (rt_c/rt_d) and the cost ratio (c_d/c_c). The response time SLO ratio represents the relationship between the current p-quantile of the response time rt_c and the target SLO response time rt_d. A ratio greater than one indicates SLO violations, signaling the need to scale up the number of replicas. Conversely, a ratio below one indicates no SLO violations, allowing for a reduction in replicas to lower infrastructure costs. The cost

[1] https://custom-pod-autoscaler.readthedocs.io

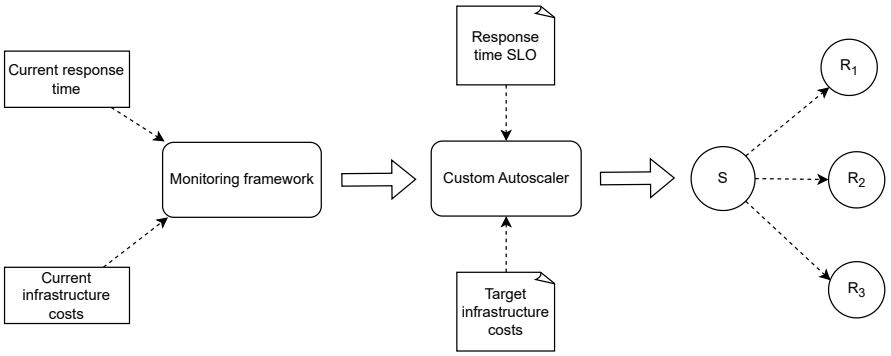

Figure 2: General model of the proposed approach.

ratio reflects the relationship between the desired target cost c_d and the currently predicted cost c_c. A ratio greater than one indicates that the current predicted costs are below the target, allowing for scaling up the number of replicas to improve performance. On the other hand, a ratio lower than one indicates predicted costs exceed the target, necessitating a scale down of replicas to reduce costs.

The w_{rt} and w_c parameters are in the range between zero and one and their sum is equal to one. By adjusting the values of the w_{rt} and w_c parameters, different weights are assigned to the response time SLO and cost ratios, respectively, in determining the value of the k parameter. A value of w_{rt} significantly higher than w_c indicates that scaling actions are primarily driven by the need to meet SLO targets. Conversely, if w_c is much higher than w_{rt}, scaling actions are primarily guided by the need to keep costs below the maximum target.

3.3 Monitoring Framework

The real-time service response times and infrastructure cost metrics are continuously collected by a comprehensive monitoring framework (Marchese and Tomarchio, 2024), as depicted in Figure 3, and are made accessible to the custom autoscaler for dynamic scaling decisions.

At the core of this monitoring framework is the Prometheus[2] metrics server, a database designed to collect, store, and query time series data. Prometheus periodically gathers metrics from various exporters and makes them available through the PromQL query language, allowing for detailed insights and real-time monitoring. The Prometheus server is deployed as a Kubernetes Deployment within the control plane.

Service response time metrics are collected using the Istio[3] framework, a service mesh implementation that manages Pod communication within the Kubernetes cluster. The Istio control plane is installed within the cluster and automatically injects a sidecar container running an Envoy proxy into each Pod upon creation. These Envoy proxies intercept all traffic between Pods, providing fine-grained observability. They expose detailed traffic statistics through metrics exporters, which can then be queried by the Prometheus server to capture real-time service response times.

Infrastructure cost metrics are collected using the OpenCost agent and node exporters. OpenCost is a vendor neutral framework designed for measuring and allocating cloud infrastructure and container costs. Specifically built for Kubernetes environments, OpenCost enables real-time cost monitoring, show back, and charge back, providing valuable insights into resource consumption and associated expenses.

The OpenCost agent, which runs as a Deployment within the Kubernetes control plane, collects node CPU and memory metrics from the Prometheus server. It then generates infrastructure cost metrics based on a a pricing model and the collected data, which are subsequently stored within Prometheus for further analysis. Node exporters, deployed as *DaemonSets* on each cluster node, continuously monitor and report CPU and memory usage, and expose those metrics to the Prometheus server.

4 EVALUATION

The proposed solution has been evaluated by using a sample application generated using the *µBench* benchmarking tool (Detti et al., 2023). µBench enables the generation of Kubernetes manifests for service-mesh topologies with one or multiple microservices, each running a specific function. Among the pre-built functions in µBench, the *Loader* function models a generic workload that stresses node re-

[2]https://prometheus.io
[3]https://istio.io

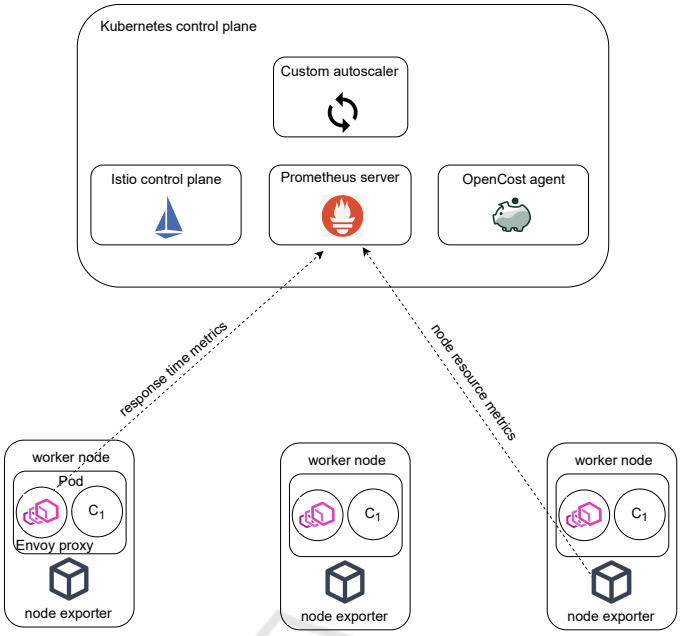

Figure 3: Monitoring framework.

sources when processing HTTP requests. When invoked, the Loader function computes an N number of decimals of π. The larger the interval, the greater the complexity and stress on the CPU. Additional stress on node memory can be configured by adjusting the amount of memory required by the function for each computation. For this work, a single-service application has been generated as a Kubernetes Deployment with resource requirements of 0.5 vCPU and 250MB of memory.

The test bed environment for the experiments consists of a Rancher Kubernetes Engine 2 (RKE2)[4] Kubernetes cluster with one master node for the control plane and a pool of worker nodes. These nodes are deployed as virtual machines on a Proxmox[5] environment and configured with 2 vCPU and 8GB of RAM. Autoscaling of worker nodes is managed by the *kproximate*[6] cluster autoscaler, which communicates with the Proxmox API server to dynamically provision and de-provision virtual machines based on the resource required by the service replicas. A pricing model that charges one unit of cost per vCPU/hour and one unit of cost per 1GB of RAM/hour is used to calculate the overall cost of provisioned cluster nodes.

Black box experiments are conducted by evaluating the end-to-end response time of the sample application and the overall infrastructure costs when HTTP requests are sent to the application service with a specified number of virtual users each sending one request every second in parallel. Requests to the application are sent through the k6 load testing utility[7] from a node inside the same network where cluster nodes are located. This setup minimizes the impact of network latency on the application response time. Each experiment consists of 10 trials, during which the k6 tool sends requests to the application for 30 minutes. For each trial, statistics about the application response time are measured and averaged with those of the other trials of the same experiment. An SLO of 300ms for the 90th of the application response time and an overall cost of 40 units both over a 30 minutes window are fixed as target values. For each experiment, we compare the performances of the proposed custom autoscaler with those of the Kubernetes Horizontal Pod Autoscaler. The custom autoscaler has a run period and stabilization window set to 30 seconds, with the autoscaling algorithm parameters w_{rt} and w_c each assigned a value of 0.5.

Figures 4 and 5 present the results of the experiments. The first graph shows the 90th percentile of the application response time in relation to the number of virtual users concurrently sending requests, while the second graph illustrates the cumulative infrastructure costs for each experiment. Across all experiments, the proposed approach consistently outperforms the Kubernetes HPA in both application response time and infrastructure costs. At lower virtual user counts, the performance of the proposed ap-

[4]https://docs.rke2.io
[5]https://www.proxmox.com
[6]https://github.com/jedrw/kproximate
[7]https://k6.io

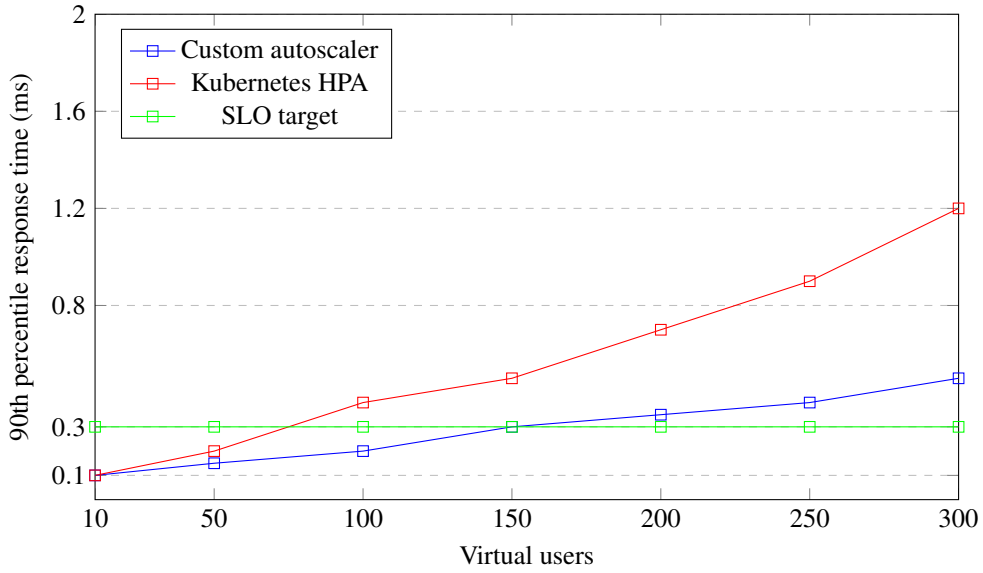

Figure 4: Service response time.

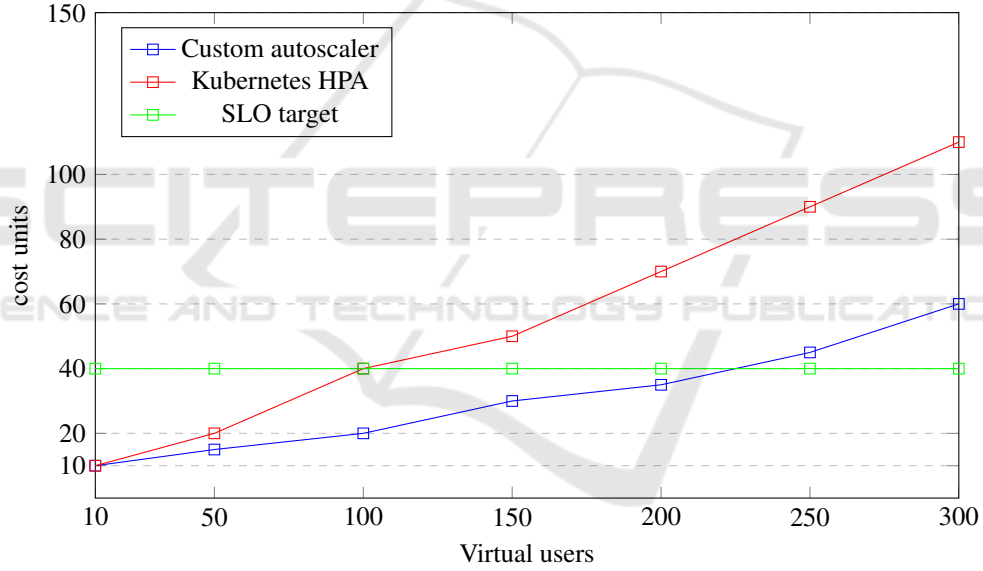

Figure 5: Infrastructure costs.

proach is similar to that of the Kubernetes HPA, as the application experiences limited load and minimal infrastructure requirements. However, as the number of virtual users increases, the proposed approach begins to significantly outperform the Kubernetes HPA, with more noticeable improvements at higher user counts. Both response time and infrastructure costs grow more rapidly with the Kubernetes HPA compared to the proposed approach.

5 RELATED WORK

In the literature, there is a variety of works that propose extensions of the Kubernetes platform in order to devise custom Pod autoscaling solutions aimed at ensure service response times while minimizing infrastructure costs (Tran et al., 2022; Do et al., 2025).

In (Marie-Magdelaine and Ahmed, 2020) authors propose a proactive autoscaling framework that uses a learning-based forecast model to dynamically adjust the resource pool, both horizontally and vertically. The framework uses a proactive autoscaling al-

gorithm based on Long Short-Term Memory (LSTM) to improve the end-to-end latency for cloud-native applications.

Libra (Balla et al., 2020) is an adaptive autoscaler, which automatically detects the optimal resource set for a single Pod, then manages the horizontal scaling process. Additionally, if the load or the underlying virtualized environment changes, Libra adapts the resource definition for the Pod and adjusts the horizontal scaling process accordingly.

In (Yuan and Liao, 2024) authors propose a predictive autoscaling Kubernetes operator based on time series forecasting algorithms, aimed to dynamically adjust the number of running instances in the cluster to optimize resource management. In this work, the Holt–Winter forecasting method and the Gated Recurrent Unit (GRU) neural network, two robust time series forecasting algorithms, are employed and dynamically managed.

Gwydion (Santos et al., 2025), is a microservices-based application autoscaler that enables different autoscaling goals through Reinforcement Learning (RL) algorithms. Gwydion is based on the OpenAI Gym library and is aimed to bridge the gap between RL and autoscaling research by training RL algorithms on real cloud environments for two opposing reward strategies: cost-aware and latency-aware. Gwydion focuses on improving resource usage and reducing the service response time by considering microservice inter dependencies when scaling horizontally.

In (Pramesti and Kistijantoro, 2022) an autoscaler based on response time prediction is proposed for microservice applications running in Kubernetes environments. The prediction function is developed using a machine learning model that features performance metrics at the microservice and node levels. The response time prediction is then used to calculate the number of Pods required by the application to meet the target response time.

StatuScale (Wen et al., 2024) is a status-aware and elastic scaling framework which is based on a load status detector that can select appropriate elastic scaling strategies for differentiated resource scheduling in vertical scaling. Additionally, StatuScale employs a horizontal scaling controller that utilizes comprehensive evaluation and resource reduction to manage the number of replicas for each microservice.

6 CONCLUSIONS

In this work, we propose extending the Kubernetes platform with a custom Pod autoscaling strategy aimed at minimizing SLO violations in the response times of containerized applications running in cloud environments, while simultaneously reducing infrastructure costs. Our primary goal is to address the limitations of the Kubernetes Horizontal Pod Autoscaler, which scales Pod replicas based on low-level resource usage metrics. This approach makes it challenging to define scaling targets that are properly correlated with the desired response time SLOs and maximum infrastructure costs. The idea is to propose a Pod autoscaling policy based on high-level metrics, such as actual application response times and infrastructure costs, to more accurately achieve the desired SLO and cost targets.

For future work, we plan to enhance the efficiency of the proposed autoscaling policy by using AI and time series analysis techniques to identify patterns in user requests and predict their trends. This will enable the development of a proactive autoscaling policy that scales up the number of replicas to ensure improved service performance, while minimizing infrastructure over provisioning and reducing unnecessary costs.

ACKNOWLEDGEMENTS

This work was partially funded by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, Mission 4 Component C2 Investment 1.1 - Call for tender No. 1409 of 14/09/2022 of Italian Ministry of University and Research - Project ”Cloud Continuum aimed at On-Demand Services in Smart Sustainable Environments” - CUP E53D23016420001.

REFERENCES

Balla, D., Simon, C., and Maliosz, M. (2020). Adaptive scaling of kubernetes pods. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5.

Calcaterra, D., Di Modica, G., Mazzaglia, P., and Tomarchio, O. (2021). TORCH: a TOSCA-Based Orchestrator of Multi-Cloud Containerised Applications. *Journal of Grid Computing*, 19(1).

Chen, T., Bahsoon, R., and Yao, X. (2018). A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Comput. Surv.*, 51(3).

Detti, A., Funari, L., and Petrucci, L. (2023). µbench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980.

Do, T. V., Do, N. H., Rotter, C., Lakshman, T., Biro, C., and Bérczes, T. (2025). Properties of horizontal pod autoscaling algorithms and application for scaling cloud-

native network functions. *IEEE Transactions on Network and Service Management*, pages 1–1.

Gannon, D., Barga, R., and Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, 4:16–21.

Gupta, H., Vahid Dastjerdi, A., Ghosh, S. K., and Buyya, R. (2017). ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296.

Hongyu, Y. and Anming, W. (2023). Migrating from monolithic applications to cloud native applications. In *2023 8th International Conference on Computer and Communication Systems (ICCCS)*, pages 775–779.

Kubernetes (2024). Production-Grade Container Orchestration. https://kubernetes.io. Last accessed 3 Jun 2024.

Marchese, A. and Tomarchio, O. (2024). Telemetry-driven microservices orchestration in cloud-edge environments. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 91–101, Shenzhen, China. IEEE Computer Society.

Marie-Magdelaine, N. and Ahmed, T. (2020). Proactive autoscaling for cloud-native applications using machine learning. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–7.

Mukherjee, A., De, D., and Buyya, R. (2024). *Cloud Computing Resource Management*, pages 17–37. Springer Nature Singapore, Singapore.

Pramesti, A. A. and Kistijantoro, A. I. (2022). Autoscaling based on response time prediction for microservice application in kubernetes. In *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, pages 1–6.

Salii, S., Ajdari, J., and Zenuni, X. (2023). Migrating to a microservice architecture: benefits and challenges. In *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*, pages 1670–1677.

Santos, J., Reppas, E., Wauters, T., Volckaert, B., and De Turck, F. (2025). Gwydion: Efficient auto-scaling for complex containerized applications in kubernetes through reinforcement learning. *Journal of Network and Computer Applications*, 234:104067.

Tran, M.-N., Vu, D.-D., and Kim, Y. (2022). A survey of autoscaling in kubernetes. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 263–265.

Wen, L., Xu, M., Gill, S. S., Hilman, M. H., Srirama, S. N., Ye, K., and Xu, C. (2024). Statuscale: Status-aware and elastic scaling strategy for microservice applications.

Yuan, H. and Liao, S. (2024). A time series-based approach to elastic kubernetes scaling. *Electronics*, 13(2).