IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received 22 November 2024, accepted 27 December 2024, date of publication 7 January 2025, date of current version 13 January 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3526751

RESEARCH ARTICLE

On the Stability of the Kubernetes Horizontal Autoscaler Control Loop

BERTA SERRACANTA[1], ANDOR LUKÁCS[2], ALBERTO RODRIGUEZ-NATAL[3], ALBERT CABELLOS[1], AND GÁBOR RÉTVÁRI[4], (Member, IEEE)
[1]Department of Computer Architecture, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain
[2]Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 400084 Cluj-Napoca, Romania
[3]Cisco, 28108 Madrid, Spain
[4]Department of Telecommunications and Artificial Intelligence, Budapest University of Technology and Economics, 1111 Budapest, Hungary

Corresponding author: Berta Serracanta (berta.serracanta@upc.edu)

This work was supported in part by the Spanish I+D+i Project TowaRds fully AI-empowered NetwoRks, subproject A (TRAINER-A), funded by Ministerio de Ciencia e Innovación (MCIN)/Agencia Estatal de Investigación (AEI)/10.13039/501100011033 under Grant PID2020-118011GB-C21; in part by the Catalan Institution for Research and Advanced Studies (ICREA) through the Secretariat for Universities and Research of the Ministry of Business and Knowledge of the Government of Catalonia; in part by the European Social Fund; and in part by the National Research, Development and Innovation Fund of Hungary under Grant OTKA/ANN-135606, Grant OTKA/FK-135074, and Grant OTKA/FK-134604.

ABSTRACT Kubernetes is a widely used platform for deploying and managing containerized applications due to its efficient elastic capabilities. The Horizontal Pod Autoscaler (HPA) in Kubernetes independently adjusts the number of pods for each service, yet these services often operate in an interconnected manner. This study aims to understand the effects of autoscaling events on a graph of interconnected services. To achieve this, we apply control theory to model the HPA's behavior. We analyze the stability of this model, perform numerical simulations, and deploy a real testbed to evaluate the performance. Our findings demonstrate that the control theory-based model accurately predicts the HPA's behavior, ensuring system stability with CPU utilization meeting desired thresholds and no traffic loss after a transitional period. The model provides insights into optimizing resource scheduling and improving application performance in Kubernetes environments. Additionally, we extend our model to the whole service graph to understand how individual scaling decisions influence the complex graphs of cloud applications.

INDEX TERMS Cloud autoscaling, control theory, Horizontal Pod Autoscaler, Kubernetes, microservices architecture, numerical simulations, system stability.

I. INTRODUCTION

Kubernetes[1] has emerged as a preferred platform for deploying and managing containerized applications, credited largely to its efficient elastic capabilities. This has allowed the platform's widespread adoption across diverse industries, with many leading organizations depending on its robust features, which is a testament to its effectiveness. It was reported in 2023 that 66% of cloud service consumers were using Kubernetes in production [1], highlighting its pervasive adoption and significant impact on the industrial landscape.

Kubernetes runs distributed applications that are typically implemented using the microservice architecture [2]. Each microservice is designed to perform a specific business function and can be developed, deployed, and scaled independently. With this, one can describe the architecture in terms of a service graph, a collection of loosely-coupled microservices. In the service graph each node is a microservice and edges exist when two nodes exchange information. In this context Kubernetes' elastic resource allocation system allocates/deallocates resources (e.g, CPU) to each node using the Horizontal Pod Autoscaling (HPA) algorithm [3].

Specifically, HPA automatically adjusts the number of pods, replicas of the same service, by continuously

The associate editor coordinating the review of this manuscript and approving it for publication was Libo Huang.

[1]Kubernetes: http://kubernetes.io/

monitoring specific metrics and scaling the service up or down. This allows dynamic scaling to match the demand, providing better application performance and resource utilization for each of the individual services in a service graph. Each service in the graph has its own independent HPA control loop, ensuring that scaling decisions are tailored to its own specific metrics.

Despite the benefits of HPA in dynamically managing resources, research on scaling complex applications following service-based architectures has highlighted several challenges. One significant issue is the lack of accurate resource estimation models, which causes current approaches to frequently involve cautious and iterative adjustments to resource allocations [4]. In this paper we present a model for Kubernetes services and use it to analyze the stability of the application service graph when operated by HPA's Kubernetes algorithm. We aim to understand if the autonomous scaling decisions made to individual services can collectively lead to a stable scaling effect for the entire application, or some unwanted effects such as large-scale oscillations emerge. Even if a large chunk of cloud applications rely on Kubernetes, and a lot of focus has been put on improving the resource allocation and performance optimization when performing autoscaling events [4], [5], little research effort has been devoted to formally model Kubernetes' HPA behavior. Our model specifically focuses on CPU-intensive applications, demonstrating that properly managing CPU bottlenecks is key to maintaining performance and ensuring system stability.

For this we employ control theory principles. This model treats each service as a plant and the HPA algorithm as the controller, with the control signal being the adjustment of the number of pods based on the CPU utilization feedback. The main contributions of this paper are (1) the formal verification of the stability of the Kubernetes HPA control loop for the single-service scenario and (2) an experimental analysis of the stability and efficiency of HPA in the multi-service scenario.

The remainder of this paper is structured as follows: Section II dives into related work, Section III presents the service model, Section IV discusses the control theory framework and stability analysis, Section V provides experimental results, and Section VI presents the conclusions.

II. RELATED WORK

In this paper, we model the functionality of the Kubernetes Horizontal Pod Autoscaler (HPA). We examine how the addition or removal of pods impacts the performance and stability of the targeted service. Once we understand how these affects a particular service, we extrapolate the findings to the entire service graph to study how an autoscaling event propagates throughout it.

There is extensive literature on models of the entire service graph. These models can be classified into two main types: white-box and black-box models, which respectively show or hide the complexities of the units being modeled. To capture all the layers involved in a microservice application, white-box models often use computationally complex solutions such as layered queuing networks [6]. Another interesting approach is to model the service mesh sidecar attached to each microservice in the graph [7] as an abstraction of the whole microservice. Black-box models, on the other hand, typically apply reinforcement learning (RL) techniques, such as GNNs or SVMs, combined with online metrics tracing [8], [9], [10]. In both cases, these models focus on applying optimization techniques directly to the modeled application, either to achieve better resource allocation guaranteeing stronger Service Level Objectives (SLO) or to run simulations in a digital twin. However, we do not intend to model all the complexities and layers of a service graph application, from the kernel to user space. Instead, we focus on modeling the behavior of the HPA in each individual microservice and their interconnections.

Rather than modeling, many studies focus on enhancing the HPA through modifications and optimizations to overcome the traditional approach of overprovisioning to avoid SLO violations. These can be classified into four different categories [4], [5] (and the references therein): (i) rule-based autoscaling, common among cloud providers; (ii) time series data analysis; (iii) queuing network models; and (iv) RL-based optimizations or a combination of several of the techniques above [11]. As pointed out in [8], many existing autoscalers manage resources for each microservice individually, which means there is a possibility of cascading effects propagated along the graph and, in turn, performance degradations. This is because these autoscalers are agnostic to changes in the workload of the graph until it reaches them directly, becoming more severe the deeper the microservice is located.

We do not aim to modify the existing autoscaler but rather seek to understand the particularities and robustness of the de facto Kubernetes HPA. This research models CPU fluctuations driving HPA autoscaling events, not only on individual microservices but across the entire graph. Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) could be used to predict these changes by forecasting incoming traffic load. However, their effectiveness is limited when clear correlations, like seasonal patterns, do not exist. To the best of our knowledge, no prior work has focused on this specific objective.

III. SERVICE MODEL

We first present the basic entities of the Kubernes autoscaling system: the pod, the service and the Horizontal Pod Autoscaler. These are the entities that we model using control theory, firstly for a single service, and then generalized to a whole service graph.

A. THE KUBERNETES HORIZONTAL AUTOSCALER

A pod is the smallest deployable unit in Kubernetes and can be thought of as a wrapper around a single container. It is deployed based on its specified resource requirements and

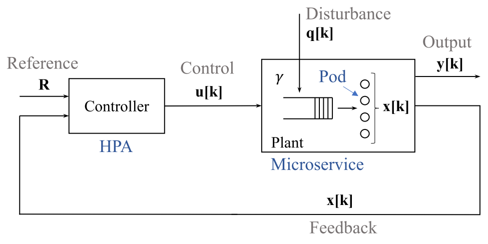

FIGURE 1. Microservice model.

TABLE 1. Variables and signals used in our model.

Variable	Description	Signal
k	Discrete time-step index	-
$q[k]$	Incoming request load (req/sec)	Disturbance
$u[k]$	CPU allocation (mcore)	Control/Input
$x[k]$	Average CPU load (mcore)	Feedback
$y[k]$	Processed requests (req/sec)	Output
R	CPU utilization threshold, $0 < R \leq 1$	Reference
γ	Scaling factor (req/sec $\rightarrow$ mcore)	-
α_0, α_1	2nd order coefficients	-
$d[k]$	Traffic loss (req/sec)	-
$G(V, E)$	Computational microservices graph	-
λ_{ij}	Request intensity from service i to j	-

limits. A microservice, also referred to as service, groups multiple pods that perform the same functions, presenting them as a single entity.

The Kubernetes Horizontal Pod Autoscaler (HPA) dynamically adjusts the number of pods allocated to a service. Each service has one dedicated HPA control loop, which manages the number of pod replicas running based on resource configuration and real-time resource consumption metrics. By default, HPA can scale pods using CPU or memory utilization metrics, with CPU usage being the most commonly used. HPA continuously monitors the service's CPU usage: if it exceeds a threshold it adds new pods, whereas if the CPU usage drops below the threshold ite HPA decreases the number of running pods to optimize resource use. Managing the number of pods running in a service can be viewed as a control system, where the service represents the plant and HPA functions as the controller, as illustrated in Figure 1.

Table 1 summarizes the notation used in the paper. Variables related to a single microservice system are denoted without a subscript (e.g., $x[k]$). Variables with the same meaning but pertaining to a specific microservice within the service graph $G(V, E)$ are denoted with a subscript i, $i \in V$ (e.g., $x_i[k]$). Where V represents the set of microservices and E the set of directed edges indicating interactions between them.

B. MICROSERVICE MODEL

Below, we provide a formal model to describe each of the components in the HPA control loop. First we introduce a model for describing the service's response to the input load in terms of CPU consumption.

When considering different approaches to modeling, it is important to account for the trade-off between the ease of analysis and the modeling capabilities and power. For example, a simple memoryless model describing how the average CPU utilization of an service $x[k]$ [percentage] varies with respect to the system's incoming load $q[k]$ [req/sec] and the total amount of CPU (number of pod replicas times CPU per pod) assigned by the HPA control loop to the service $u[k]$ [mcore], would be the following:

$$x[k+1] = \frac{\gamma(q[k])}{u[k]} \tag{1}$$

Here, $\gamma(.)$ [mcore/req/sec] is a generic function that describes the ideal CPU requirement the service needs to process a given load. In general, γ may be linear (see below), it may describe a diminishing returns characteristics often found in practice [12], or it may be any monotonically increasing function. The essence of the microservice model is then that the average CPU utilization $x[k]$ in the k-th timestep equals the total amount of CPU $\gamma(q[k])$ required to serve the current load $q[k]$ divided by the total CPU $u[k]$ assigned by the HPA controller.

Unfortunately, this model, while formally analyzable, is memoryless (i.e., the output depends only on the state at the current timestep k). Thus, it does not account for critical parameters shaping system dynamics, like the impact of queue buffering (which is dependent on the state in *earlier* timesteps). Therefore, below we use a slightly more complex non-linear recursive model inspired by Finite Impulse Response (FIR) systems in control theory [13].

Equation 2 gives the general form of the microservice model used throughout this paper. Here, N denotes the order of the system and α_n are the coefficients that weigh the contribution of CPU utilization in the previous timestep $k-n$ to the current state. For brevity, we assume that $\gamma(.)$ is linear henceforth.

$$x[k+1] = \gamma \sum_{n=0}^{N} \left(\frac{\alpha_n q[k-n]}{u[k-n]} \right) \tag{2}$$

In the following we assume a minimum threshold for the number of CPU units of 1 assigned per each service. This means that each service retains at least one pod, even in the absence of incoming traffic. This assumption is based on the practical need to ensure service availability and responsiveness.

C. CONTROLLER MODEL

Next, we formally model the Kubernetes Horizontal Pod Autoscaler (HPA) control loop. The HPA control loop aims to maintain the CPU utilization around a target level by continuously adjusting the number of pods based on the CPU utilization measured from the service. If the CPU utilization exceeds a given upper threshold then HPA assigns more pods to the service. Conversely, if the CPU utilization

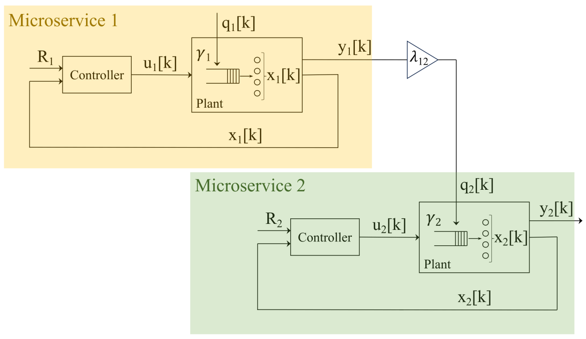

FIGURE 2. **Service graph model as a 2-service chain.**

is below a lower threshold, HPA reduces the number of pods to optimize resource usage. This closed-loop system ensures that the microservice adapts to varying loads while maintaining performance and resource efficiency.

In our model we represent the upper and the lower thresholds with a single reference threshold R (a user-configurable parameter). Our HPA model is then a direct formal representation of the HPA control law specified in the official Kubernetes documentation [14]:

$$u[k+1] = u[k]\frac{x[k+1]}{R} \tag{3}$$

Here, $u[k]$ [mcore] denotes the control input, which determines the total CPU units assigned to the microservice.

We quantify the amount of traffic $y[k]$ at the service output in response to a given input load and the available compute resources as follows:

$$y[k+1] = \min(q[k], \frac{q[k]}{x[k]}) \tag{4}$$

D. SERVICE GRAPH MODEL

Next we extend the model to a conceptual application constituted by *multiple* linked microservices, as illustrated in Figure 2. This composite model supposes a distinct plant and an HPA control loop for each microservice. These elements form a microservice graph $G(V, E)$, where G represents a rooted Directed Acyclic Graph (DAG) with the root denoted by $r \in V$.

In this model, the instantaneous propagation of traffic across microservices can be expressed as in Equation 5, which takes into account that the system can not process more traffic than the input load.

$$q_j[k] = \lambda_{ij} y_i[k] \qquad (i, j) \in V \tag{5}$$

Here, $\lambda_{ij} : (i, j) \in V$ defines the intensity of requests transmitted from microservice i to j.

E. TRAFFIC LOSS

The stability of stability of the proposed model will be evaluated based on a *traffic loss* metric, which describes the resource deficit of the current CPU allocation. The traffic loss metric $d_i[k]$ quantifies the amount of requests the application is unable to process due to the lack of available CPU resources:

$$d_i[k] = q_i[k] - y_i[k] \tag{6}$$

In general, $d_i[k] = 0$ means a perfect CPU allocation, while $d_i[k] > 0$ indicates service degradation and/or interruption, affecting the overall application performance. The target of the HPA control loop is to drive the system to a state where the loss is zero.

Similarly, the aggregated traffic loss metric for the service graph model, represented by the set V, is computed as the sum of each of the loss at each service at a given time step k: $d[k] = \sum_{i \in V} d_i[k]$. Here, $d_i[k]$ is the single service quality metric as previously defined.

IV. STABILITY ANALYSIS

In this section, we present the main contribution of the paper: a formal stability analysis of the single-service HPA control loop in terms of the traffic loss metric. In the subsequent section we extend the analysis to the multi-service model using numerical evaluations.

First, we introduce a simplification. In particular, since our approach is highly non-linear we reduce the general microservice model to a second-order model. This means that the CPU dynamics is fully described by the coefficients α_0 and α_1, plus the input $q[.]$ and $u[.]$ This simplification allows us to incorporate both the current CPU value and the value from the previous time step into the model, providing a balance between model accuracy and ease of analysis. The proposed control model has a time complexity of $O(N^2)$, where N represents the evaluated time steps.

Theorem 1: Consider the system described by the second-order application dynamics (2), the HPA control law (3), and let the input be a step function:

$$q[k] = \begin{cases} 1 & \text{if } k \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Assume $x[k] \in (0, +\infty)$ and $u[k] \in (0, +\infty)$, $u[0] = 1$. Then, the control system is globally asymptotically stable, i.e.:

1) The steady state error is zero: $\lim_{k\to\infty} x[k] - R = 0$.
2) The steady state traffic loss is zero: $\lim_{k\to\infty} d[k] = 0$.

These conditions define stability as a system that allocates the exact amount of resources required to meet the corresponding load by aligning the CPU consumption with the desired threshold R. When there is a mismatch between the incoming load and the allocated resources, the system compensates by either adding or removing pods ensuring that in steady state there are the necessary resources.

In order to demonstrate the previous stability definitions we start by analyzing the system boundaries and convergence of the control signal $u[k]$ and later extrapolate the results to find the steady-state error and traffic loss.

Proof: Using $B = \frac{\gamma q[k]}{R}$ to define $b_0 = B\alpha_0$ and $b_1 = B\alpha_1$. Let $b_0, b_1, u[0], u[1] \in (0, +\infty)$ and define the sequence $(u[k])_{k\in\mathbb{N}}$ by the recursion:

$$u[k+2] = b_0 + b_1 \frac{u[k+1]}{u[k]}, \quad \forall k \in \mathbb{N} \tag{7}$$

Then $(u[k])_{k\in\mathbb{N}}$ converges to $b_0 + b_1$.

Using the transformation $u[k] = b_0(1+z[k])$, the recursion becomes:

$$\begin{aligned} b_0(1+z[k+2]) &= b_0 + b_1 \frac{1+z[k+1]}{1+z[k]}, \\ 1+z[k+2] &= 1 + \frac{b_1}{b_0}\frac{1+z[k+1]}{1+z[k]}, \\ z[k+2] &= \frac{b_1}{b_0}\frac{1+z[k+1]}{1+z[k]}, \quad \forall k \in \mathbb{N}. \end{aligned} \tag{8}$$

Next, we introduce a Lemma from prior work that we use to establish the main result:

Lemma 1 (Theorem 6.3.3 of [15]): Let us consider the variables $p, q, z_0, z_1 \in (0, +\infty)$ and define the sequence:

$$z[k+2] = \frac{p + qz[k+1]}{1+z[k]}, \quad \forall k \in \mathbb{N}. \tag{9}$$

The unique positive equilibrium of this recursion is globally asymptotically stable if one of the following two conditions holds:

1) $q < 1$;
2) $q \geq 1$ and either $p \leq q$ or $q < p \leq 2(q+1)$.

Using the notations of Lemma 1, we have $p = q = \frac{b_1}{b_0}$. The case $b_0 > b_1$ translates to $q < 1$, and the case $b_0 \leq b_1$ translates to $q \geq 1$ and $p = q$. Thus, the sequence $(z[k])_{k\in\mathbb{N}}$ converges for every $b_0, b_1, z_0, z_1 \in (0, +\infty)$ to the unique positive equilibrium of the recursion which is $\frac{b_1}{b_0}$. This means that the sequence $(u[k])_{k\in\mathbb{N}}$ converges to $b_0 + b_1$.

Finally, by undoing the substitutions we obtain that $(u[k])_{k\in\mathbb{N}}$ converges to $\frac{\gamma q}{R}(\alpha_0 + \alpha_1)$. In turn, the control signal makes the plant $(x[k])_{k\in\mathbb{N}}$ converge at the desired steady-state:

$$x[k] = \gamma \left(\frac{\alpha_0 R q[k]}{\gamma q[k](\alpha_0+\alpha_1)} + \frac{\alpha_1 R q[k-1]}{\gamma q[k-1](\alpha_0+\alpha_1)} \right) = R \tag{10}$$

And consequently, we can demonstrate that in steady state we achieve null traffic loss with the following simple computation:

$$\begin{aligned} \lim_{k\to\infty} d[k] &= \lim_{k\to\infty} (q[k] - y[k]) \\ &= Q - \lim_{k\to\infty} \min(q[k], \frac{q[k]}{x[k]}) = 0 \end{aligned} \tag{11}$$

Corollary 1: Consider an application graph $G(V, E)$ composed of autonomous services operating independently. If every service in the graph fulfills the requirements in Theorem 1, then the service graph G is globally asymptotically stable.

This corollary follows directly from Theorem 1. If every autonomous service in the graph G satisfies the conditions, then each service is stable, exhibits no deviation error, and has no traffic loss in steady state. Since these services operate independently, their stability ensures the overall stability of the graph G.

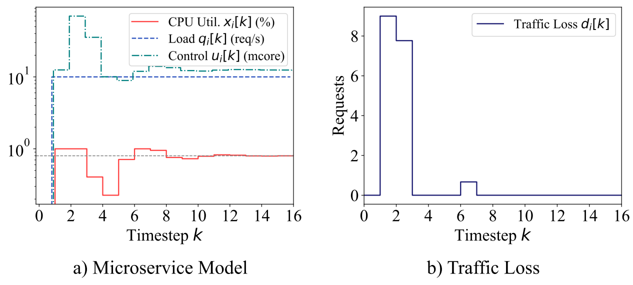

FIGURE 3. Microservice model performance.

V. RESULTS

To further illustrate the stability demonstrated through the analytical solutions, this section presents the results obtained from both numerical simulations and real deployment experiments. These results provide graphical insights that validate the theoretical findings on HPA stability.

A. NUMERICAL SIMULATIONS

We have conducted numerical simulations using Matlab Simulink [16] for both the microservice model, with time complexity $O(N!)$, and the service graph model. These simulations aim to visualize the stability behavior and dynamic response of the Kubernetes HPA under various conditions, providing a controlled environment to verify our theoretical analysis.

Figure 3 shows the main signals of the microservice model, the incoming load $q[k]$ (blue line), the CPU utilization $x[k]$ (red) and the control signal $u[k]$ (green line) discussed in the previous sections for a user-defined HPA threshold of $R = 0, 8$, meaning that in steady state we desire an average CPU utilization of $x[k] = 0, 8$. It can be seen that given an input load the system reacts and starts autoscaling by allocating more CPU resources and reducing them until the desired steady state is achieved at $k = 14$.

It is important to note that typical values for the CPU-targeted HPA threshold are around 30%, as applications in production cannot afford to lose traffic and typically overprovision to prevent this. However, for verification purposes, we use a threshold of $R = 0.8$ (80%) CPU consumption to observe the model's behavior under more stressful conditions.

When we extend these results to the service graph, as depicted in Fig. 4, we observe that the stability achieved by each independent and stable service is reflected in the overall system behavior. As shown, for each of the three independent services deployed in a chain-like configuration the CPU utilization follows a similar pattern to the results achieved in the microservice model simulations, meaning that all three of them are able to independently adjust their resources to reach a steady state without being impacted by the autoscaling decisions of other nodes in the graph.

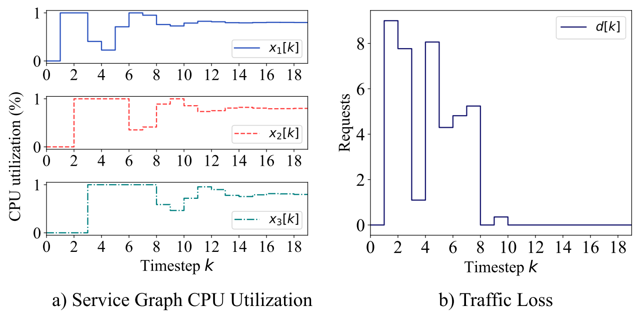

FIGURE 4. **Service graph model performance.**

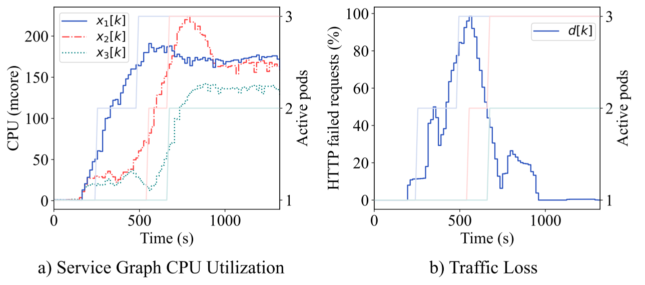

FIGURE 5. **Kubernetes 3-Service chain deployment performance.**

In summary, these results demonstrate that the entire system can reach the desired steady state after a transitional period, where CPU utilization aligns with the target threshold and traffic loss is eliminated.

B. EXPERIMENTAL RESULTS

To verify our theoretical analysis in a real-world scenario, we deployed a Kubernetes testbed focusing on the general case of the service graph model for a CPU-intensive application. This real deployment aims to observe the HPA's performance and stability in a practical setting, ensuring that the analytical and simulation results hold true in live environments.

The experimental setup comprises three interconnected microservices configured in a cascading topology, as detailed in [17]. Each service is CPU load generator that, upon receiving a request, executes arithmetic operations for 8ms. The request is then forwarded to the subsequent service in the sequence.

We deployed Kubernetes' HPA (v2) on each microservice. HPA is configured to auto-scale based on CPU consumption and configured to trigger an autoscaling event to deploy another replica when CPU utilization reaches 80% of the requested 100mcore per service. This mimics the previous analysis for direct comparison.

To generate the requests we use K6 [18], a well-established synthetic load generator. Specifically we define an initial virtual spike of 20 users that plateaus at 22 virtual users. In total each experiments lasts 20 minutes.

Figure 5 illustrates the CPU usage for each microservice involved in our system. The first microservice initially receives the incoming load, leading to an early rise in its CPU consumption. As this service processes the traffic, the load is then transferred to the second microservice, which exhibits a similar increase in CPU usage. Eventually, the load reaches the third and final microservice. The vertical lines represent the active pods for each service, highlighting the points at which CPU usage at each service meets the Horizontal Pod Autoscaler (HPA) threshold, prompting an autoscaling event to deploy a new replica.

It demonstrates the impact of CPU saturation on service performance. Until a new replica is successfully deployed, the affected service's CPU remains saturated, making it unable of processing additional incoming requests. This results in the loss of traffic. Notably, the request failures start when autoscaling events are triggered. These losses remain until the deployment of additional replicas sufficiently increases resource availability, allowing the system to manage the incoming load effectively.

These experimental results validate the theoretical model and testbed configuration. They demonstrate that the system effectively reaches the desired steady state where CPU utilization meets the threshold and traffic loss is minimized after the initial transitory period. This confirms the practical applicability and reliability of the HPA under real-world conditions.

VI. CONCLUSION

Kubernetes, with its extensive deployment over the past several years, has proven to be a crucial platform for managing containerized applications in large-scale production environments [1]. Many organizations depend on Kubernetes for its robust and dynamic scaling capabilities, underscoring its importance and effectiveness.

This work not only models and validates the stability of HPA in both single and multi-service scenarios but also lays the groundwork for future advancements in cloud application scaling and resource management. This can further enhance application performance in Kubernetes environments, i.e., by optimizing resource allocation and scheduling, which is critical given the platform's widespread industrial adoption.

In conclusion, our research builds on the extensive use and proven success of Kubernetes and HPA in production environments. By addressing the analytical gaps, we provide initial steps towards a more in-depth study and open up research possibilities in optimizing Kubernetes' scaling mechanisms.

ACKNOWLEDGMENT

The authors would like to express their gratitude to the editor and the reviewers for their valuable comments and constructive feedback. Their insightful suggestions have greatly contributed to the improvement of this manuscript. They deeply appreciate their time and effort in reviewing their work.

REFERENCES

[1] Cloud Native Computing Foundation. (2023). *2023 Annual Survey*. Accessed: May 29, 2024. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/

[2] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "The evolution of distributed systems towards microservices architecture," in *Proc. 11th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2016, pp. 318–325.

[3] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*, 3rd ed., New York, NY, USA: O'Reilly Media, 2022.

[4] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–33, Jul. 2018. [Online]. Available: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3148149

[5] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, Dec. 2014.
C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–33, Jul. 2018. [Online]. Available: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3148149

[6] E. Incerto, R. Pizziol, and M. Tribastone, "μOpt: An efficient optimal autoscaler for microservice applications," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. (ACSOS)*, Sep. 2023, pp. 67–76.

[7] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, "Dissecting overheads of service mesh sidecars," in *Proc. ACM Symp. Cloud Comput.*, New York, NY, USA, Oct. 2023, pp. 142–157, doi: 10.1145/3620678.3624652.

[8] J. Park, B. Choi, C. Lee, and D. Han, "GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, 2021, pp. 154–167, doi: 10.1145/3485983.3494866.

[9] H. Qiu, S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Aug. 2020, pp. 805–825. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[10] D. Borsatti, W. Cerroni, L. Foschini, G. Ya Grabarnik, L. Manca, F. Poltronieri, D. Scotece, L. Shwartz, C. Stefanelli, M. Tortonesi, and M. Zaccarini, "KubeTwin: A digital twin framework for Kubernetes deployments at scale," *IEEE Trans. Netw. Service Manage.*, vol. 21, no. 4, pp. 3889–3903, Aug. 2024.

[11] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1994–2004.

[12] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[13] A. Oppenheim, A. Willsky, and I. Young, *Signals and Systems* (Prentice-Hall Signal Processing Series). Upper Saddle River, NJ, USA: Prentice-Hall, 1983. [Online]. Available: https://books.google.es/books?id=UQJRAAAAMAAJ

[14] *Horizontal Pod Autoscaling: Algorithm Details*. Accessed: Jun. 28, 2024. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[15] M. Kulenovic and G. Ladas, *Dynamics of Second Order Rational Difference Equations: With Open Problems and Conjectures*, 1st ed., Boca Raton, FL, USA: CRC Press, 2001, doi: 10.1201/9781420035384.

[16] MathWorks, Inc., Natick, MA, USA. (2022). *Matlab Version: 9.13.0 (r2023a)*. [Online]. Available: https://www.mathworks.com

[17] *Kubernetes Horizontal Autoscaling Benchmark*. Accessed: May 2, 2024. [Online]. Available: https://github.com/rg0now/k8s-hpa-benchmark/tree/main?tab=readme-ov-file

[18] *K6*. Accessed: May 2, 2024. [Online]. Available: https://k6.io/docs/

BERTA SERRACANTA is currently pursuing the Ph.D. degree with UPC BarcelonaTech. Her research focuses on network-enabled application acceleration, exploring the integration of network and application layers, and the optimization of distributed systems for enhanced operational efficiency.

ANDOR LUKÁCS received the Ph.D. degree in mathematics from Utrecht University. He is currently a Lecturer with Babeş-Bolyai University. His research interests include abstract homotopy theory, operads, dendroidal sets, and metric fixed point theory.

ALBERTO RODRIGUEZ-NATAL received the Ph.D. degree from BarcelonaTech, with a thesis on software-defined networking. He is a Senior Technology Lead with the Enterprise Networking CTO Team, Cisco, where he works in the intersection of network and applications.

ALBERT CABELLOS received the Ph.D. degree in 2008. He has been a Full Professor with the Computer Architecture Department, Universitat Politècnica de Catalunya, since 2020. He is the co-founder of Barcelona Neural Networking (https://bnn.upc.edu/) and the NaNoNetworking Center in Catalunya (https://www.n3cat.upc.edu/).

GÁBOR RÉTVÁRI (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from Budapest University of Technology and Economics (BME), and the D.Sc. degree from the Hungarian Academy of Sciences. He is currently an Associate Systems Professor with the Department of Telecommunications and Artificial Intelligence, BME. He is interested in all theoretical and practical aspects of distributed systems and data networking.