6-2024

Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures

Hussain AHMAD

Christoph TREUDE
Singapore Management University, ctreude@smu.edu.sg

Markus WAGNER

Claudia SZABO

Citation

AHMAD, Hussain; TREUDE, Christoph; WAGNER, Markus; and SZABO, Claudia. Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures. (2024). *Proceedings of the 21st International Conference on Software Architecture, Charminar, Hyderabad, India, 2024 June 4-8*. 1-12.
Available at: https://ink.library.smu.edu.sg/sis_research/8961

Smart HPA: A Resource-Efficient Horizontal Pod Auto-scaler for Microservice Architectures

Hussain Ahmad*, Christoph Treude†, Markus Wagner§, Claudia Szabo*
*University of Adelaide, Australia, †University of Melbourne, Australia, §Monash University, Australia
{hussain.ahmad, claudia.szabo}@adelaide.edu.au, christoph.treude@unimelb.edu.au, markus.wagner@monash.edu

***Abstract*—Microservice architectures have gained prominence in both academia and industry, offering enhanced agility, reusability, and scalability. To simplify scaling operations in microservice architectures, container orchestration platforms such as Kubernetes feature Horizontal Pod Auto-scalers (HPAs) designed to adjust the resources of microservices to accommodate fluctuating workloads. However, existing HPAs are not suitable for resource-constrained environments, as they make scaling decisions based on the individual resource capacities of microservices, leading to service unavailability and performance degradation. Furthermore, HPA architectures exhibit several issues, including inefficient data processing and a lack of coordinated scaling operations. To address these concerns, we propose Smart HPA, a flexible resource-efficient horizontal pod auto-scaler. It features a hierarchical architecture that integrates both centralized and decentralized architectural styles to leverage their respective strengths while addressing their limitations. We introduce resource-efficient heuristics that empower Smart HPA to exchange resources among microservices, facilitating effective auto-scaling of microservices in resource-constrained environments. Our experimental results show that Smart HPA outperforms the Kubernetes baseline HPA by reducing resource overutilization, overprovisioning, and underprovisioning while increasing resource allocation to microservice applications.**

***Index Terms*—Microservices, Auto-scaling, Self-Adaptation, Software Architecture, Resource Management, Kubernetes**

I. Introduction

Microservice architectures have gained widespread popularity in recent years [1], with several prominent enterprises (e.g., Netflix, Amazon, and Spotify) and defense systems transitioning from monolithic to microservices for improving their service quality [2], [3]. In the microservice software design paradigm, complex applications are broken down into a set of autonomous, loosely coupled, and fine-grained services called microservices. Each microservice has a defined functionality and communicates with others through lightweight interfaces such as the HTTP resource API [1]. Microservice architectures accelerate application delivery and improve reliability, as each microservice is designed, developed, and deployed independently [4]. In general, microservices are deployed using software containers, with container orchestration platforms being widely adopted for the runtime management of microservices [2]. While various container orchestration platforms such as Kubernetes [5], Docker Swarm [6], and Red Hat OpenShift [7], are available [8], Kubernetes is the most widely adopted platform in both academia and industry [9].

Auto-scaling is an essential requirement for microservice architectures as it allows a microservice to dynamically adjust its computing resources to handle fluctuating workloads (e.g., Slashdot effect [10]) without human intervention [11]. For instance, during holiday seasons such as Black Friday, websites like Amazon experience a tenfold increase in workload compared to their usual levels [12]. This increased load depletes the allocated resources for active microservices, resulting in longer response times and service unavailability. To address this challenge, container orchestration platforms include a Horizontal Pod Auto-scaler (HPA) that automatically adjusts the number of replica pods within a microservice deployment in response to fluctuating workloads [13].

HPAs consider workload fluctuation, resource utilization, and user requirements specified in a Service-Level Agreement (SLA) [14] to determine the required number of microservice replicas in dynamic environments. However, existing HPAs have a number of limitations, including the inability to scale a microservice deployment beyond the predefined resource limits [13], and the vulnerability of their architectures to failures [9]. In the context of microservice deployment, each microservice within an application is allocated specific resources (i.e., maximum replica limit) [11], [13] and HPAs are bound by these predefined maximum replica limits. Hence, when a microservice application experiences a high workload, HPAs are unable to scale busy microservices within the application beyond their maximum replica limits [13]. This limitation, in turn, leads to service unavailability, performance degradation, and financial losses [15]. At the same time, less busy microservices in the application have surplus resources, leading to resource wastage and additional operational costs [16]. In addition, HPA control architectures have been implemented either in a centralized [17] or a decentralized [18] manner, resulting in a single point of failure [19] or a lack of coordination among scaling decisions [9], respectively. Recent improvements to decentralized solutions have been proposed through master-worker [20] and hierarchical [21] architectures. However, these improvements have not effectively addressed the communication overhead challenges [22].

To address these challenges, we propose Smart HPA as an extension of Kubernetes. Smart HPA incorporates a hierarchical architecture that combines centralized and decentralized architectural styles. The decentralized architecture of Smart HPA consists of dedicated auto-scalers for each microservice that handle scaling operations for their respective microservices. The centralized component of Smart HPA activates only in resource-constrained scenarios, facilitating resource exchange among microservices to enable resource-efficient scaling of microservices. Through this deliberate restriction of communication interactions between decentralized and centralized

The paper has been accepted at the 21st IEEE International Conference on Software Architecture (ICSA) 2024

components, Smart HPA reduces communication overhead compared to traditional hybrid approaches, as the extent of communication overhead depends on these interactions [22]. Furthermore, through the utilization of decentralized auto-scalers, the proposed hierarchical architecture mitigates challenges associated with centralized architectures, such as the vulnerability of a single point of failure. In addition, Smart HPA provides flexibility regarding the scaling policies employed by dedicated auto-scalers and the scaling metrics utilized, such as CPU usage and response time, for executing auto-scaling of microservices. This flexibility caters to the needs of researchers and practitioners by enabling them to tailor scaling policies and metrics according to their requirements. In summary, our contributions are three-fold.

- We propose a hierarchical architecture that integrates both centralized and decentralized architectural styles to capitalize on their advantages while addressing limitations in managing auto-scaling operations for microservice applications.
- We develop resource-efficient heuristics that transfer resources from overprovisioned to underprovisioned microservices within a microservice application, facilitating scaling operations in resource-constrained environments.
- We evaluate the performance of Smart HPA against the widely used Kubernetes HPA using a real-world microservice benchmark application. With the default configurations of the benchmark application, our experimental results show that Smart HPA excels with 5x less overutilization, 7x lower overprovisioning, no underprovisioning, and a 1.8x boost in microservice resource allocation.

We provide the replication package [23] for Smart HPA, encompassing all scripts and data essential for reproducing, validating, and extending the results outlined in the paper.

II. Background and Related Work

In this section, we investigate existing scaling policies and architectural designs of HPAs.

A. Scaling Policies

i) Threshold-based scaling policy. Threshold-based policies are widely used in both academia (e.g., [18], [24]–[26]) and industrial container orchestration platforms (e.g., Kubernetes, Amazon ECS) to determine desired replica counts of microservices using different scaling metrics such as CPU and memory usage [27]. A threshold-based scaling policy is composed of conditions that trigger scaling actions, such as "If the CPU usage exceeds 80 percent, then scale up the microservice". While threshold-based policies are straightforward in concept, they require manual threshold adjustment for dynamic auto-scaling of microservices [16]. Furthermore, defining threshold-based policies for complex microservice infrastructures with resource contention and inter-dependencies is challenging [21].
ii) Fuzzy-based scaling policy. A fuzzy-based scaling policy refers to a predetermined set of "If-Else" rules that rely on human knowledge of microservice applications to make scaling decisions [28]–[30]. Fuzzy inference, in contrast to threshold-based policies, permits the use of descriptive terms (e.g., high, medium, low) rather than exact numbers when specifying rules for scaling decisions [31]. Fuzzy-based policies, like threshold-based scaling, demand a comprehensive understanding of the application for defining fuzzy rules [16], [21].
iii) Queuing theory-based scaling policy. The existing literature (e.g., [9], [17], [32], [33]) highlights the use of queuing theory to estimate scaling metrics (i.e., response time) for different workload levels. In queuing theory, a microservice application is represented as a queuing model, where both service and inter-arrival times follow general statistical distributions [2]. However, the accuracy of queuing models can be compromised when there are significant deviations from the exponential distribution in inter-arrival or service times [9], [34]. Also, queuing models provide approximate estimations, and fine-tuning their parameters requires thorough application profiling, which can be costly and time-consuming [21].
iv) Control theory-based scaling policy. A control theory-based scaling policy [15], [35]–[37] fine-tunes system behavior by evaluating the current value of a scaling metric against its reference value within a feedback loop. For example, Baarzi et al. [15] proposed a proportional–integral–derivative controller *SHOWAR* that leverages the history of scaling decisions and current scaling metrics to formulate the next auto-scaling decisions. Nevertheless, these policies can be time-consuming during their decision-making process, particularly when scaling metrics interact in a complex way because of the inter-dependencies among microservices [2].
v) Artificial Intelligence-based scaling policy. An Artificial Intelligence (AI) based scaling policy utilizes Machine Learning (ML) and Reinforcement Learning (RL) models to estimate scaling metrics for executing auto-scaling of microservices [38]. For regression-based auto-scalers [39]–[41], ML models use historical data of scaling metrics (e.g., CPU and memory usage) for decision-making. However, frequent changes in workload patterns can lead to high costs and time for model retraining [2]. For RL-based auto-scalers, RL agents determine scaling actions by engaging in a sequence of interactions with their environment [27]. Model-free RL algorithms, such as Q-learning and SARSA [42]–[44], suffer from slow learning rates. This results in time-consuming auto-scaling of microservices [9], [16]. While model-based RL approaches [21], [45], [46] can address the slow convergence issue of model-free methods, they face issues related to scalability when applied in large-scale microservice architectures [2].

B. Control Architectures

Control architecture refers to the structural design of an auto-scaler that is responsible for executing scaling operations based on a scaling policy within microservice applications. In general, control architectures are implemented through MAPE-K control loop components, involving Monitoring, Analysis, Planning, and Execution of auto-scaling operations, facilitated by a Knowledge Base [47]. In the existing literature, we have identified two distinct architectural styles for auto-scalers: (i) centralized, and (ii) decentralized.

Centralized architecture. The majority of the existing HPAs, such as [17], [48], [49], follow a centralized architectural style for the execution of scaling operations within microservice applications. In this architecture, all data generated by microservices within a microservice application is collected and processed in a central auto-scaler. This central auto-scaler is then responsible for formulating and implementing scaling decisions for all microservices. Although the design of centralized architectures is simple, the centralization of data management leads to several challenges, including the risk of a single point of failure, limited scalability, longer processing times, and a heavier computational load [9].

Decentralized architecture. To address the limitations of centralized architectures, in decentralized architectural style [18], [43], each microservice within a microservice application is equipped with an independent, dedicated auto-scaler. These individual auto-scalers are responsible for collecting and processing data from their respective microservice and making and executing scaling decisions exclusively for those specific microservices. Moreover, most of the industrial container orchestration platforms (e.g., Kubernetes, Amazon ECS) employ fully decentralized architectures for auto-scaling operations [13]. However, the lack of synchronization among decentralized auto-scalers can lead to frequent scaling operations, particularly in scenarios involving interdependent microservices that contend for resources [9]. This situation degrades the performance of microservice applications. To address this issue, recent works have proposed hierarchical [9], [21] and master-worker [19], [20], [45] decentralized architectural styles that make coordination among independent auto-scalers. However, these approaches introduce communication overhead and bottleneck problems due to increased communication between workers and master auto-scalers during auto-scaling operations [22].

Distinguishing features of Smart HPA. The architecture of Smart HPA incorporates both centralized and decentralized components to leverage the strengths of both architectural styles while mitigating their limitations. In resource-rich environments, Smart HPA follows a fully decentralized architectural style, wherein each microservice is equipped with its dedicated auto-scaler, to avoid the limitations of centralized architectures, such as a single point of failure. However, in scenarios where resources are limited, Smart HPA employs a hybrid approach, where it uses a centralized adaptation module for resource management along with decentralized auto-scalers to ensure coordination. It is important to highlight that, unlike existing hierarchical and master-worker architectures, Smart HPA mitigates communication overheads and bottleneck issues by utilizing centralized data processing capabilities only when necessary. Besides, Smart HPA can seamlessly incorporate any scaling policy within its dedicated auto-scalers. While existing auto-scaling policies are bound to make scaling decisions according to the pre-defined resource limits of microservice deployments, Smart HPA enables scaling policies to execute decisions that go beyond the resource limits of microservice deployments. As a prototype implementation, we show the benefits of Smart HPA instantiated with a threshold-based scaling policy due to its straightforward implementation.

III. Smart HPA Architecture

In this section, we describe the hierarchical architecture and resource-efficient heuristics of Smart HPA. Fig. 1 presents the hierarchical architectural design of our proposed Smart HPA, which consists of three main components: *Microservice Manager*, *Microservice Capacity Analyzer*, and *Adaptive Resource Manager*. To adapt to frequent resource congestion, Smart HPA incorporates the components of the MAPE-K framework [47] into both the decentralized Microservice Managers and the centralized Adaptive Resource Manager for executing the auto-scaling of microservices.

As presented in Fig. 1, Smart HPA employs a dedicated Microservice Manager for each microservice within a microservice application running on a Kubernetes cluster. This dedicated allocation of Microservice Managers offers a high degree of flexibility in auto-scaling operations. For instance, it allows the customization of scaling policies, goals, and metrics tailored to the requirements of each microservice within an application. This provides a clear separation of adaptation goals among individual microservices. Moreover, all Microservice Managers operate in a fully decentralized manner, working in parallel to collect and process data. This decentralized architecture results in enhanced monitoring and improved time efficiency, as opposed to a sequential centralized approach. Initially, all Microservice Managers collect and analyze data from their respective microservices, using a scaling policy to make scaling decisions based on the requirements of each microservice. Subsequently, the Microservice Capacity Analyzer assesses the feasibility of executing scaling decisions by comparing resource demands and resource capacities of all microservices within an application. In resource-constrained situations, where the resource demand exceeds the capacity of a microservice within an application, the Microservice Capacity Analyzer triggers the intervention of the centralized Adaptive Resource Manager. This manager employs resource-efficient heuristics, coordinating Microservice Managers to exchange resources among microservices, and facilitating the formulation and execution of scaling decisions for each microservice.

It is noteworthy that data generated by the components of Smart HPA for each microservice, such as resource utilization and scaling decisions, is stored within the Knowledge Base of the Smart HPA architecture. The Knowledge Base facilitates further data processing within Smart HPA and enhances situational awareness for key stakeholders, such as developers, practitioners, and users. In the following, we discuss the components of Smart HPA in detail.

A. Microservice Manager

The Microservice Manager is composed of the components of the MAPE-K framework [47]. Its primary role involves the collection and analysis of data from a microservice to determine its desired replica count for making a scaling decision accordingly. The Execute component of a Microservice Manager scales a microservice based on instructions from either the Microservice Capacity Analyzer or the Adaptive Resource

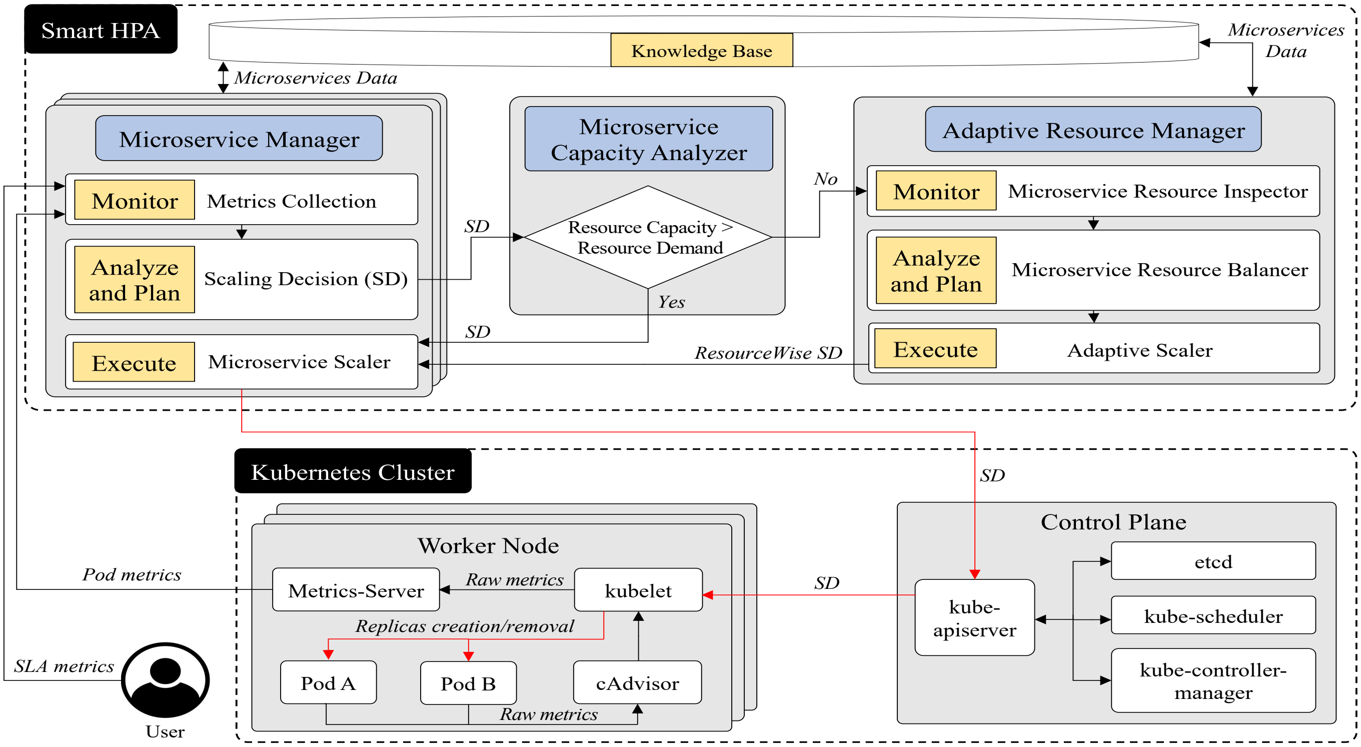


Fig. 1: Proposed Hierarchical Architecture of Smart HPA.

Manager (Fig. 1). Algorithm 1 outlines the functionality of a Microservice Manager dedicated to a microservice *i*.

1) Monitor: The Monitor component is responsible for collecting data pertaining to a specific microservice. To make effective scaling decisions, the Monitor component collects essential pod and SLA metrics associated with a microservice. Pod metrics refer to key parameters that reflect a microservice's current status, such as resource utilization (*CMV*) and current replica count (*CR*). Similarly, SLA metrics provide insights into user requirements, including resource threshold value (*TMV*) and minimum/maximum replica counts (*minR, maxR*) of a microservice. Once the data collection phase is complete, the collected metrics are then forwarded to the Analyze component for subsequent data processing.

2) Analyze and Plan: The Analyze and Plan component is responsible for determining the desired replica counts, identifying violations, and making scaling decisions for a microservice. The functionality of the Analyze and Plan component for a microservice *i* is provided in Algorithm 1 and summarized in the following steps:

Step 1: Desired Replicas Determination. We employ a static threshold-based scaling policy, similar to the one used in Kubernetes HPA [13], within the Analyze and Plan components of all Microservice Managers to determine desired replica counts for their respective microservices. In Algorithm 1, the first line illustrates the threshold-based scaling policy, which factors in the current replica count (CR_i), resource metric utilization (CMV_i), and resource threshold value (TMV_i) of a microservice *i* to compute its desired replica count.

Algorithm 1 MICROSERVICE MANAGER

CMV: current value of resource metric, *CR:* current replica count, *DR:* desired replica count, *TMV:* threshold value of resource metric, *maxR:* maximum replica count, *minR:* minimum replica count, *SD:* scaling decision, *KB:* knowledge base

```
Monitor: $CMV_i, TMV_i, CR_i, minR_i, maxR_i$
Output: $DR_i, SD_i$
 1: $DR_i = \text{ceil}(CR_i \times (CMV_i/TMV_i))$
 2: if $DR_i > CR_i$ then
 3:     $SD_i = $ Scale up
 4: else if $DR_i < CR_i$ and $DR_i \geq minR_i$ then
 5:     $SD_i = $ Scale down
 6: else
 7:     $SD_i = $ No Scale
 8: end if
 9: $KB \leftarrow DR_i, SD_i, CMV_i, TMV_i, CR_i, minR_i, maxR_i$
10: return $DR_i, SD_i, maxR_i$
```

Step 2: Violation Detection. Violations occur when a desired replica count DR_i for microservice *i* is not equal to its current replica count CR_i. This indicates the current resource utilization of a microservice is not within its threshold limit. Lines 2 and 4 of Algorithm 1 specify the violation detection conditions and highlight the need for auto-scaling operations.
Step 3: Scaling Decision. Upon detecting a violation, an adaptation process is triggered. This leads to subsequent scale-up or scale-down operations for a microservice. Lines 2-7 of Algorithm 1 outline the scaling decisions *SD* corresponding to different types of detected violations.

Following the scaling decision, the Microservice Manager stores the processed data related to its microservice in the Knowledge Base *KB* (line 9). Moreover, it transmits the scaling decision *SD*, desired replica count *DR*, and maximum replica count *maxR* to the Microservice Capacity Analyzer (line 10).

B. Microservice Capacity Analyzer

When an adaptation is triggered by a Microservice Manager, Smart HPA needs to create or remove replicas for the respective microservice in accordance with its scaling decision (*SD*). As discussed, each microservice within an application is allocated defined resources [11]. It is critical to determine whether the resource demand (*DR*) falls within the bounds of the resource capacity (*maxR*) of a microservice. Therefore, we introduce the Microservice Capacity Analyzer, dedicated to assessing the resource demand and resource capacity for each microservice within a microservice application.

As shown in Fig. 1, the Microservice Capacity Analyzer first gathers information on the resource demand and resource capacity for each microservice. Subsequently, in case the resource demand for each microservice aligns with its resource capacity (i.e., $DR_i \leq maxR_i$), the Microservice Capacity Analyzer instructs the Execute components of Microservice Managers to proceed with the scaling decisions for their respective microservices. Alternatively, when any microservice, within an application, demands more resources than its resource capacity (i.e., $DR_i > maxR_i$), the Microservice Capacity Analyzer activates the centralized component of Smart HPA (i.e., Adaptive Resource Manager) to exchange resources among microservices for executing scaling decisions. This purposeful activation of the centralized Adaptive Resource Manager allows Smart HPA to potentially reduce communication overhead between its centralized and decentralized components.

C. Adaptive Resource Manager

The centralized component of our hierarchical Smart HPA, the Adaptive Resource Manager, establishes coordination among decentralized Microservice Managers to execute scaling decisions in resource-constrained environments (i.e., $DR_i > maxR_i$). It optimizes resource exchange among microservices to ensure that each microservice's resource demand is adequately met, all while considering the overall resource capacity available for the whole application. We propose resource-efficient heuristics outlined in Algorithm 2 for the Adaptive Resource Manager to enable resource-wise scaling of microservices. Fig. 1 presents the MAPE-K components of the Adaptive Resource Manager, which are distributed across three key components: Microservice Resource Inspector, Microservice Resource Balancer, and Adaptive Scaler.

1) Microservice Resource Inspector: Lines 1-14 of our heuristics presented in Algorithm 2 provide operational details of the Microservice Resource Inspector. To efficiently exchange resources among microservices, it is crucial to determine which microservices have residual resources and which ones are in need of additional resources. Therefore, the Microservice Resource Inspector identifies over- and under- provisioned microservices within a microservice application. It calculates required resources *Underprov* for underprovisioned microservices (lines 4-7) and residual resources *Overprov* for overprovisioned microservices (lines 8-11).

Algorithm 2 ADAPTIVE RESOURCE MANAGER

Overprov: overprovisioned microservices' resource, *Underprov:* underprovisioned microservices' resource, *ResReq:* resource request value for a replica, *ResSD:* resource-wise scaling decision, *ResDR:* resource-wise desired replica count, *TotalR:* total replica count from residual resources

```
Input: $DR_i, SD_i, maxR_i, ResReq_i$                    $\forall i = 1, \cdots, M$
Output: $ResSD_i, ResDR_i$                               $\forall i = 1, \cdots, M$
// Calculating residual and required resources for all services
 1: function MICROSERVICE RESOURCE INSPECTOR
 2: $Overprov$ =[ ], $Underprov$ =[ ]
 3: for $i = 1$ to $M$ do      // M = total number of microservices
 4:     if $DR_i > maxR_i$ then
 5:         $RequiredR_i = DR_i - maxR_i$
 6:         $RequiredRes_i = RequiredR_i \times ResReq_i$
 7:         $Underprov$.append($RequiredRes_i$)
 8:     else
 9:         $ResidualR_i = maxR_i - DR_i$
10:         $ResidualRes_i = ResidualR_i \times ResReq_i$
11:         $Overprov$.append($ResidualRes_i$)
12:     end if
13: end for
14: return $Underprov, Overprov$
    // Resource transfer from overprovision to underprovision
    services; suggesting feasible desired replicas (FeasibleR)
    and resource capacities (UmaxR) for all services
15: function MICROSERVICE RESOURCE BALANCER
16: $FeasibleR$ =[ ], $UmaxR$ = [ ]
17: $U$ = length($Underprov$), $O$ = length($Overprov$)
18: $TotalOverprov = sum(Overprov)$ // total residual resource
    // Resource reallocation for underprovisioned microservices
19: $Underprov \leftarrow$ Dsort($Underprov$) // sort in descending order
20: for $i = 1$ to $U$ do    // $U$ = total underprovision microservices
21:     $TotalR_i = TotalOverprov / ResReq_i$
22:     if $TotalR_i \geq RequiredR_i$ then
23:         $FeasibleR_i, UmaxR_i = DR_i$
24:     else if $TotalR_i \in [1, RequiredR_i)$ then
25:         $FeasibleR_i, UmaxR_i$ = floor($TotalR_i$) + $maxR_i$
26:     else
27:         $FeasibleR_i$, $UmaxR_i = maxR_i$
28:     end if
29:     $UsedRes_i = (FeasibleR_i - maxR_i) \times ResReq_i$
30:     $TotalOverprov = TotalOverprov - UsedRes_i$
31: end for
    // Resource reallocation for overprovisioned microservices
32: $Overprov \leftarrow$ Asort($Overprov$) // sort in ascending order
33: for $i = 1$ to $O$ do // $O$ = total overprovisioned microservices
34:     $TotalR_i = TotalOverprov / ResReq_i$
35:     if $TotalR_i \geq ResidualR_i$ then
36:         $UmaxR_i = maxR_i$
37:     else if $TotalR_i \in [1, ResidualR_i)$ then
38:         $UmaxR_i$ = floor($TotalR_i$) + $DR_i$
39:     else
40:         $UmaxR_i = DR_i$
41:     end if
42:     $FeasibleR_i = DR_i$
43:     $UsedRes_i = (maxR_i - UmaxR_i) \times ResReq_i$
44:     $TotalOverprov = TotalOverprov - UsedRes_i$
45: end for
46: return $FeasibleR, UmaxR$
    // Updating desired replica counts, scaling decisions, and
    resource capacities for all microservices
47: function ADAPTIVE SCALER
48: for $i = 1$ to $M$ do
49:     if $FeasibleR_i == DR_i$ then
50:         $ResSD_i = SD_i$
51:     else if $FeasibleR_i \in (maxR_i, DR_i)$ then
52:         $ResSD_i$ = Scale up
53:     else
54:         $ResSD_i$ = No Scale
55:     end if
56:     $maxR_i = UmaxR_i$, $ResDR_i = FeasibleR_i$
57: end for
58: $KB \leftarrow Underprov, Overprov, ResSD_i, ResDR_i, maxR_i$
59: return $ResSD_i, ResDR_i$
end
```

2) Microservice Resource Balancer: The Microservice Resource Balancer transfers resources from overprovisioned to underprovisioned microservices, leading to potential changes in resource capacities (*maxR*) and desired replica counts (*DR*) of microservices. Consequently, the Microservice Resource Balancer suggests feasible desired replica counts (*FeasibleR*) and updated resource capacities (*UmaxR*) for all microservices. The proposed heuristics for the Microservice Resource Balancer are presented in lines 15-46 of Algorithm 2.

To prioritize addressing the needs of highly underprovisioned microservices (i.e., those experiencing a high load), the Microservice Resource Balancer initiates a process by extracting residual resources from the most heavily overprovisioned microservice, typically the one with the greatest residual resources, and reallocates these resources to the most underprovisioned microservice. Therefore, we sort the underprovisioned microservices' resource (*Underprov*) in descending order (line 19), and the overprovisioned microservices' resource (*Overprov*) in ascending order (line 32). This process iteratively proceeds, starting from the most severely underprovisioned microservice and gradually addressing less underprovisioned ones, until the resource requirements of all underprovisioned microservices are fulfilled (lines 20-31). Consequently, the Microservice Resource Balancer reduces the resource capacities of overprovisioned microservices, indicating retrieval of residual resources from them (lines 33-45). In cases where residual resources from overprovisioned microservices are insufficient to address the demands of underprovisioned microservices, the Microservice Resource Balancer determines the maximum possible desired replica count for underprovisioned microservices (lines 24-25). This is achieved by utilizing the remaining residual resources, thereby maximizing the use of overprovisioned resources to cater to the most pressing needs of underprovisioned microservices. Moreover, in highly resource-constrained situations, where no residual resources are available, no resource exchange takes place (lines 26-27).

3) Adaptive Scaler: Once the Microservice Resource Balancer suggests feasible replica counts (*FeasibleR*) and updated resource capacities (*UmaxR*) for all microservices, the Adaptive Scaler makes scaling decisions and changes resource capacities and desired replica counts accordingly for each microservice within an application (lines 48-57 of Algorithm 2). Here, we denote scaling decisions as resource-wise scaling decisions *ResSD* and desired replica counts as resource-wise desired replica counts *ResDR*. Subsequently, the Adaptive Scaler communicates the *ResSD* and *ResDR* for all microservices to the respective Execute components of Microservice Managers for implementing scaling decisions on corresponding microservices (Fig. 1). Moreover, the Adaptive Scaler stores all data, including *maxR*, *ResSD*, *ResDR*, *Underprov*, and *Overprov*, in the Knowledge Base *KB* of Smart HPA (line 58).

It is important to mention that our proposed heuristics (Algorithm 2) can integrate with any scaling policy (Section II), and metrics, such as CPU usage and response time. This flexibility allows researchers and practitioners to easily choose scaling policies and metrics according to specific requirements.

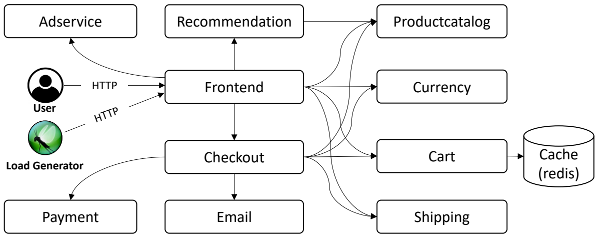


Fig. 2: Online Boutique Architecture [52].

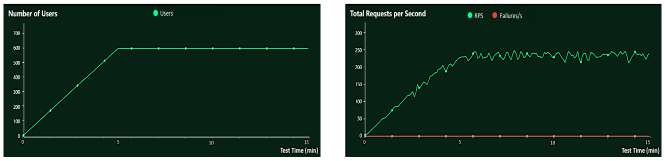
IV. Experimental Evaluation

A. Experimental Setup

Experiment Environment. We use Amazon Web Services (AWS) [50] to assess the performance of Smart HPA. We deploy 10 virtual machines (VMs) to host a benchmark microservice application. Each VM is configured as an Amazon Elastic Compute Cloud (Amazon EC2) *t3.medium* instance, equipped with an Intel Xeon Platinum 8000 series processor, 2-core 3.1 GHz CPU, 4GB of RAM, 5Gbps network bandwidth, 5GiB disk size, supporting up to 3 elastic network interfaces and 18 IP addresses. These instances run on the Linux operating system (AL2_×86_64) with the EKS-optimized Amazon Linux AMI. We use Amazon Elastic Kubernetes Service (EKS) [51] to deploy a benchmark microservice application. The Amazon EKS cluster employs Kubernetes version 1.24, with default AWS VPC network and subnet settings, IPv4 IP cluster family, API server endpoints having both private and public network access, and incorporates add-ons networking features of EKS cluster, such as kube-proxy, CoreDNS, and VPC CNI. Smart HPA is hosted on a local machine, featuring an Intel Corei7 2.60GHz CPU and 16GB RAM. Smart HPA is connected to the application running on AWS EKS through the AWS command-line interface.

Benchmark Microservice Application. Smart HPA can seamlessly integrate with any microservice application running on a Kubernetes cluster, highlighting its flexibility across diverse microservice applications. To evaluate Smart HPA, we use a real-world microservice benchmark application, Online Boutique [52], as it conforms to the benchmark selection criteria detailed in [53] and has been widely adopted by the research community, contributing to advancing the state-of-the-art in microservice architectures [27], [54], [55]. Online Boutique is a web-based e-commerce application that allows users to browse products, add items to their shopping carts, and make purchases. The application comprises 11 microservices, implemented in various programming languages.The application also provides a load test script that enables us to assess the scalability of the microservice architectures on which it is deployed. To expedite the deployment process and reduce network bandwidth usage, we have pre-downloaded all the Docker images associated with the Online Boutique onto each VM.

Application Load Testing. As described earlier, the Online Boutique application provides a load test script to simulate end users for analyzing its scalability. The script simulates user interactions with the benchmark application, such as visiting the



(a) Simulated Users. (b) Simulated Workload.

Fig. 3: Load Test for Benchmark Application.

TABLE I: Evaluation Metrics.

Evaluation Metric	Description
Supply CPU (milliCPU)	CPU resource of current replicas allocated to a microservice.
CPU Overutilization (percent usage)	CPU utilization of a microservice exceeding a predefined threshold value.
Overutilization Time (minutes)	Total duration during which at least one microservice undergoes CPU overutilization.
CPU Overprovision (milliCPU)	The residual CPU resource not utilized by a microservice, (CPU capacity - CPU demand).
Overprovision Time (minutes)	Total duration during which no microservice operates with insufficient CPU resource.
CPU Underprovision (milliCPU)	The CPU resource that a microservice needs but is unavailable, (CPU demand - CPU capacity).
Underprovision Time (minutes)	Total duration during which at least one microservice operates with insufficient CPU resource.

homepage, browsing products, adding items to the cart, viewing the cart, checking out, and setting the currency. To execute the load test script, we employ the Locust [56] load testing tool. We run the Locust on the same local machine where Smart HPA is deployed. As shown in Fig. 2, Locust sends HTTP requests to the benchmark application hosted on AWS EKS. Fig. 3 illustrates how Locust simulates users (Fig. 3a) and the associated workload (Fig. 3b) on the benchmark application. The load test is configured to run for a total duration of 15 minutes. In the initial 5 minutes, the test starts with 0 users and gradually increases, simulating the addition of 600 concurrent users with a 2-second spawn rate. This initial phase serves to analyze the behavior of Smart HPA against increasing resource demand. Following this, there are 10 minutes of sustained high load, where all 600 concurrent users actively simulate HTTP requests. This sustained high load creates a resource-constrained scenario for Smart HPA.

Evaluation Metrics. Existing literature has explored a range of resource metrics for executing horizontal pod auto-scaling in microservice architectures. These metrics include, but are not limited to response time, throughput, CPU utilization, traffic load, and memory usage. In this study, we use *CPU utilization* as the scaling metric, aligning with the default metric used by Kubernetes baseline HPA. The common scaling policy and metric selection between Smart HPA and Kubernetes HPA form a solid basis for comparing the two auto-scalers. To assess the hierarchical architecture and resource-efficient heuristics of Smart HPA, we require evaluation metrics that offer insights into resource capacity, demand, shortage, and residual aspects. Therefore, we focus on metrics related to CPU resources outlined in Table I. For example, CPU Underprovision provides insights into required CPU resources, while CPU Overprovision details residual CPU resources. Further details on the evaluation metrics are provided in Table I. It is important to note that Smart HPA offers flexibility to work with other scaling policies and metrics. For instance, if we opt to use response time as a scaling metric and implement a corresponding scaling policy (e.g., [9]) in Microservice Managers, Smart HPA can effectively carry out scaling operations with its resource-efficient heuristics reported in Section III-C.

Experimental Scenarios. As discussed in Section I, the performance of HPAs is influenced by resource threshold configurations and resource capacities of microservices.

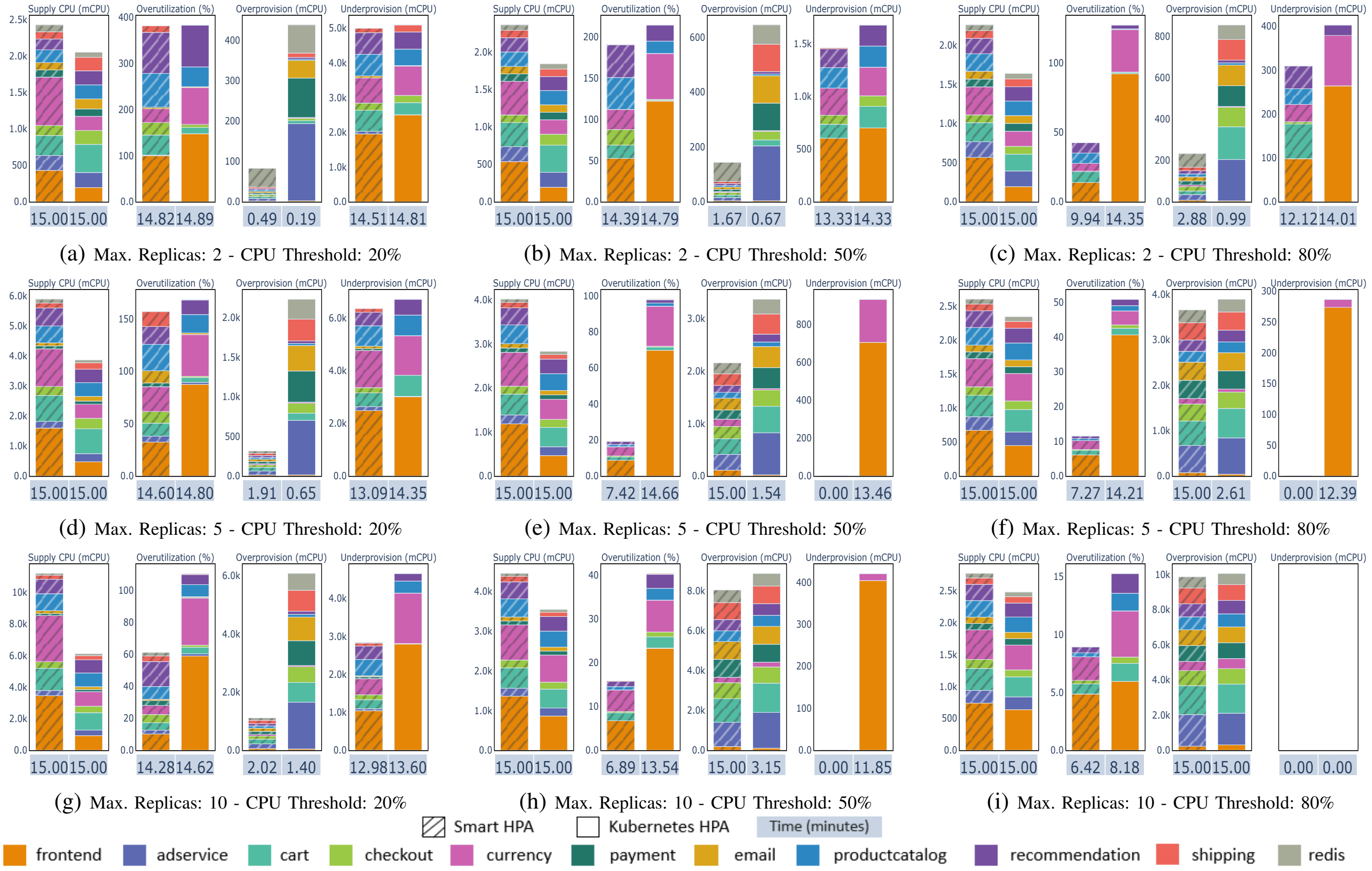
Therefore, to determine the effectiveness of Smart HPA, we have designed experimental scenarios that cover a range of resource capacities and resource threshold configurations. For resource capacities, we change the maximum number of replicas for each microservice in the benchmark application to simulate various resource-constrained levels for Smart HPA. Specifically, we set resource capacities at 2, 5, and 10 replica counts per microservice. Within each of these settings, we introduce variations in CPU threshold configurations. In particular, we experiment with CPU threshold configurations set at 20%, 50%, and 80%. This variation in resource capacities and threshold configurations across individual microservices within the benchmark application creates a total of 9 distinct experimental scenarios. We denote each experimental scenario using a combination of the threshold and maximum replica count. For example, we denote the experimental scenario featuring 10 replicas and 50% CPU Utilization as 10R-50%.

B. Results and Discussion

We compare the performance of Smart HPA against the Kubernetes baseline HPA. For each experimental scenario, we execute our load test on the benchmark application with Smart HPA and Kubernetes HPA separately. To ensure the reliability of our findings, we conduct each load test 10 times for every scenario and subsequently calculate average results. The benchmark application is configured with default settings, where each replica of all microservices has a CPU request of 100m and a CPU limit of 200m, except for *adservice* and *cart* service, which have a CPU request of 200m and a CPU limit of 300m, and *redis* service with a CPU request of 70m and a CPU limit of 125m. In Fig. 4, we present the performance evaluation results for Smart HPA and Kubernetes HPA across all experimental scenarios. The figure depicts results at both the application level and microservice level, with different colors representing the 11 microservices within the benchmark application. We observe that Smart HPA consistently outperforms Kubernetes HPA across all resource levels, ranging from 2 to 10 replicas, and threshold settings spanning from 20% to 80% CPU utilization.

Smart HPA vs. Kubernetes HPA: Analyzing Optimal and Challenging Scenarios. We analyze the most and least favorable scenarios for Smart HPA compared to Kubernetes HPA in terms of evaluation metrics.

Overutilization and Underprovision CPU and Time. In the scenario 5R-50% (Fig. 4e), Smart HPA shows no CPU Underprovision, while Kubernetes HPA records 934.04m CPU Underprovision for 13.46 minutes. This marks the highest improvement in both CPU Underprovision and Time compared to all nine scenarios, attributed to the efficient CPU resource



(a) Max. Replicas: 2 - CPU Threshold: 20%

(b) Max. Replicas: 2 - CPU Threshold: 50%

(c) Max. Replicas: 2 - CPU Threshold: 80%

(d) Max. Replicas: 5 - CPU Threshold: 20%

(e) Max. Replicas: 5 - CPU Threshold: 50%

(f) Max. Replicas: 5 - CPU Threshold: 80%

(g) Max. Replicas: 10 - CPU Threshold: 20%

(h) Max. Replicas: 10 - CPU Threshold: 50%

(i) Max. Replicas: 10 - CPU Threshold: 80%

Fig. 4: Results for all Experimental Scenarios: Lower values, better performance (except Supply CPU and Overprovision Time).

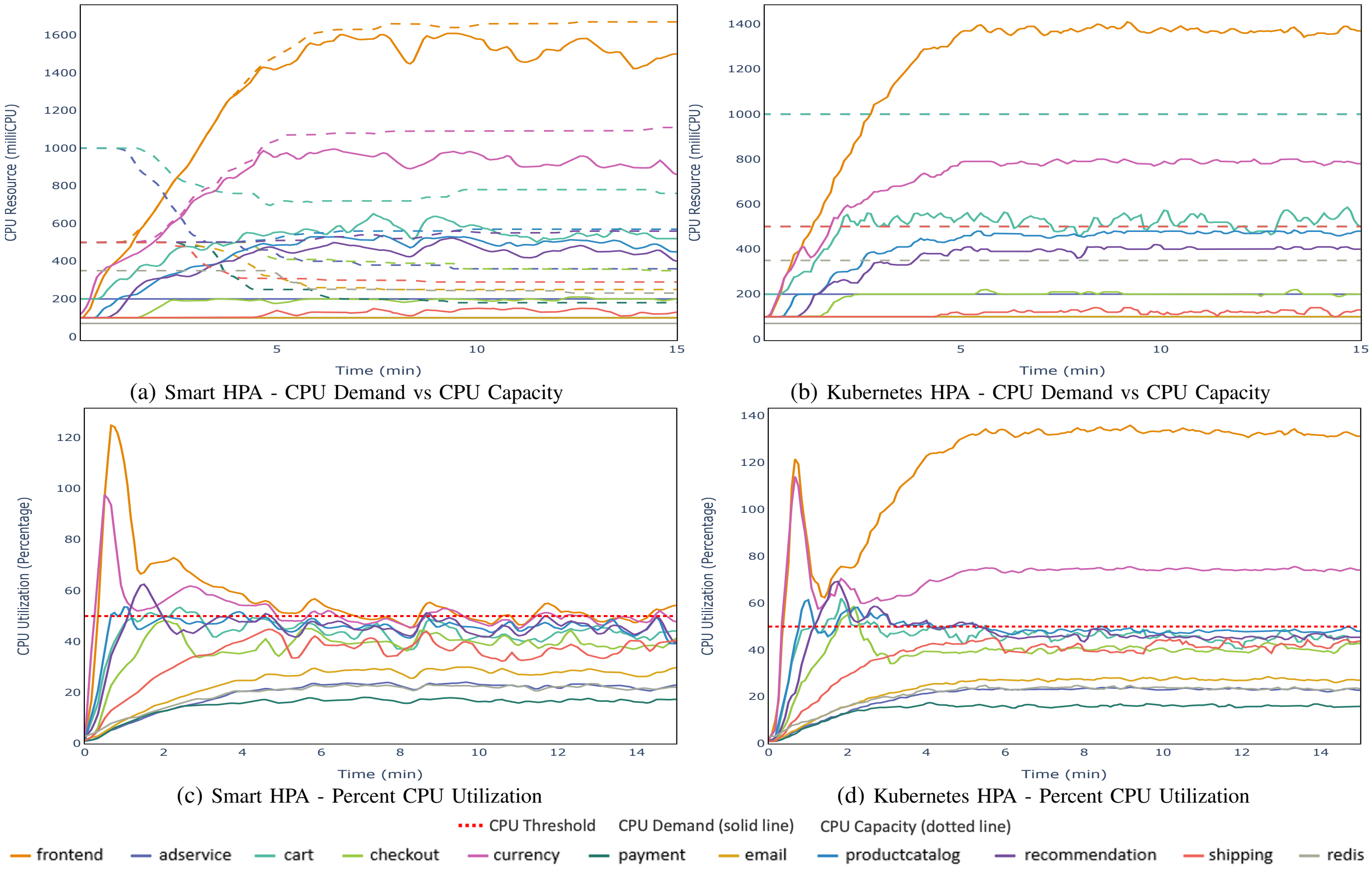
balancing of Smart HPA. Consequently, Smart HPA results in 19.30% CPU Overutilization for 7.42 minutes, compared to Kubernetes HPA's 98.06% CPU Overutilization for 14.66 minutes, indicating a significant 5.08x reduction in CPU Overutilization and a 1.98x decrease in Overutilization Time compared to Kubernetes HPA. Conversely, in the scenario, 2R-20% (Fig. 4a), where a lower CPU threshold and fewer replicas contribute to increased resource scarcity, Smart HPA demonstrates only a marginal 1.004x reduction in CPU Overutilization and Time, and a 1.01x reduction in CPU Underprovision and Time compared to Kubernetes HPA.

Overprovision CPU and Time. Low CPU thresholds enable auto-scalers to rapidly scale microservices to their maximum replica counts during high workloads. This gives Smart HPA a performance advantage in reducing CPU Overprovision by exchanging resources among microservices to align the resource capacities of microservices with their resource demands. For instance, in scenarios 2R-20%, 5R-20%, and 10R-20%, Smart HPA demonstrates 5.29x, 7.07x, and 5.46x reduction in CPU Overprovision, respectively, compared to Kubernetes HPA. Regarding Overprovision Time, Smart HPA significantly outperforms Kubernetes HPA in scenarios where the benchmark application experiences zero CPU Underprovision. For example, in scenarios 5R-50%, 5R-80%, and 10R-50%, Smart HPA exhibits 9.74x, 5.75x, and 4.76x increase in Overprovision Time, respectively, compared to Kubernetes HPA. On the other hand, in scenario 10R-80% (Fig. 4i), where no microservice encounters CPU underprovisioning under both Smart HPA and Kubernetes HPA, Smart HPA exhibits only a minimal 1.01x reduction in CPU Overprovision and no improvement in Overprovision Time compared to Kubernetes HPA.

Supply CPU. Smart HPA allocates the residual capacity of overprovisioned microservices to underprovisioned ones, resulting in a higher supply of CPU resources compared to Kubernetes HPA to the benchmark application. With a low CPU threshold triggering auto-scalers to generate more microservice replicas (i.e., Supply CPU) in response to high workloads, in scenario 10R-20% (Fig. 4g), Smart HPA supplies 1.83x more CPU resources (11188.76m) to the benchmark application compared to the 6110.41m Supply CPU provided by Kubernetes HPA. Conversely, in scenario 10R-80% (Fig. 4i), where no CPU underprovisioning occurs, Smart HPA holds a slight performance edge over Kubernetes HPA, supplying 1.11x more CPU resources (2771.42m) compared to 2478.62m CPU resource provided by Kubernetes HPA to the benchmark application during the 15-minute load test.

In summary, Smart HPA's dynamic architecture allows it to outperform Kubernetes HPA under high workload scenarios.

Smart HPA vs. Kubernetes HPA: Adaptive Behaviour Comparison. To understand the comparative behavior of Smart



(a) Smart HPA - CPU Demand vs CPU Capacity

(b) Kubernetes HPA - CPU Demand vs CPU Capacity

(c) Smart HPA - Percent CPU Utilization

(d) Kubernetes HPA - Percent CPU Utilization

Fig. 5: Smart HPA vs. Kubernetes HPA: CPU Utilization, CPU Demand, and CPU Capacity for the Scenario 5R-50%.

HPA and Kubernetes HPA during the load test, we present an entire scenario 5R-50% in Fig. 5. The figure depicts the evolution in CPU utilization, CPU demand, and CPU capacity for each microservice throughout the load test. At the start of the test, we observe that microservices start using CPU resources as the workload increases, as depicted in Fig. 5c and 5d. The rise in CPU utilization of microservices leads to an increase in their CPU demand, a pattern observed for both Smart HPA (Fig. 5a) and Kubernetes HPA (Fig. 5b).

In the case of Smart HPA, as shown in Fig. 5a, around 1.50 minutes into the test, the CPU demand for the *frontend* exceeds its allocated CPU capacity of 500m. At this point, the Microservice Capacity Analyzer triggers the Adaptive Resource Manager. The manager identifies the *adservice* as the most overprovisioned, with 1000m CPU resources, and transfers some of its CPU resources to address the underprovisioned state of the *frontend*, ensuring that its capacity matches its demand. As a result, the capacity of the *frontend* increases to meet its increasing demand, while the capacity of the *adservice* decreases but remains above its demand (Fig. 5a). Similarly, when the *currency* experiences a resource shortage around 2 minutes into the test, the Adaptive Resource Manager allocates the necessary CPU resources from the most overprovisioned *cart* microservice. When overprovisioned microservices like *adservice* and *cart* can no longer provide additional resources due to their capacity approaching their demand, the Adaptive Resource Manager reallocates CPU resources from other overprovisioned microservices, such as *email* and *shipping*, to address the underprovisioned state of the *frontend* and *currency*. In this way, Smart HPA effectively prevents microservices from becoming underprovisioned. Consequently, as illustrated in Fig. 5c, CPU utilization of both the *frontend* and *currency* experiences a decline and maintains a closer proximity to the 50% CPU threshold value.

In contrast, Kubernetes HPA does not facilitate resource sharing among microservices (Fig. 5b). This results in constant capacity levels (i.e., 500m and 1000m) for microservices throughout the load test. Consequently, this leads to a shortfall of CPU resources for the *frontend* and *currency*. Hence, during high workload, the CPU utilization of both the *frontend* and *currency* remains around 130% and 70%, respectively, surpassing the 50% CPU threshold value (Fig. 5d). Therefore, when we compare Smart HPA to Kubernetes HPA (Fig. 4e), Smart HPA significantly reduces CPU Overutilization and Overutilization Time by 5.08x and 1.98x, respectively. It also decreases CPU Overprovision by 1.56x and increases Overprovision Time by 9.74x compared to Kubernetes HPA. Notably, Smart HPA experiences no CPU Underprovision, while Kubernetes HPA encounters 934.04m CPU Underprovision for 13.46 minutes. Thus, Smart HPA's resource exchange prevents microservices from underprovisioning, resulting in superior performance across all metrics compared to Kubernetes HPA.

Implications of Variable Resource Settings.

Implications of changing CPU threshold settings. We notice a consistent pattern in our evaluation metrics for both Smart HPA and Kubernetes HPA with changing CPU threshold configurations (i,e., 20%, 50%, and 80%), regardless of the number of replicas allocated to microservices. As illustrated in Fig. 4, we observe a decrease in Supply CPU as the threshold value rises. For instance, in scenarios 5R-20%, 5R-50%, and 5R-80%, Smart HPA decreases the Supply CPU from 5887.53m to 4013.82m and further to 2601.18m, respectively. This suggests that, as the CPU threshold increases, microservices operate effectively with fewer replicas. Furthermore, with increasing CPU threshold values, Fig. 4 demonstrates a decreasing trend in CPU Overutilization, CPU Underprovision, and their respective time metrics, accompanied by an increase in CPU Overprovision and its associated time metric. This is due to the fact that each replica of microservice deployments provides more CPU capacity as the CPU threshold increases, which results in reduced CPU Overutilization and CPU Underprovision, and an increase in CPU Overprovision.

Implications of changing resource capacities. With an increasing number of replicas (i.e., 2, 5, and 10), Fig. 4 exhibits an increase in Supply CPU, CPU Overprovision, and Overprovision Time for both Smart HPA and Kubernetes HPA. This increase results from the additional CPU capacity stemming from the increased number of replicas. For example, in scenarios 2R-20%, 5R-20%, and 10R-20%, Smart HPA exhibits a rising trend in CPU Overprovision, with corresponding values of 82.67m, 315.40m, and 1110.91m (Fig. 4). As a result of this increased CPU capacity, we observe a decrease in CPU Overutilization, CPU Underprovision, and their associated time metrics.

Implications of changing both CPU threshold and resource capacities. When we investigate the combined impact of altering both threshold configurations and the number of replicas in microservice deployments, we notice that the Supply CPU tends to be higher when the threshold is low and the number of replicas is high. For instance, as illustrated in Fig. 4, scenario 10R-20% records the highest Supply CPU of 11188.76m for Smart HPA and 6110.41m for Kubernetes HPA among all scenarios. This occurs because a low CPU threshold triggers auto-scalers to generate more microservice replicas in reaction to a high workload. Moreover, as both the CPU threshold and the number of replicas increase, CPU capacity also rises, leading to higher CPU Overprovision and Time metrics. Simultaneously, this leads to reduced CPU Overutilization, Underprovision, and their respective Time metrics. Therefore, scenario 10R-80% has the highest CPU Overprovision (9864.39m) and Overprovision Time (15 minutes) and the lowest CPU Overutilization (8.93%) and Overutilization Time (6.42 minutes), with zero CPU Underprovision and Time for Smart HPA.

In summary, increasing CPU threshold or capacity for microservices decreases CPU overutilization and underprovisioning, while increasing CPU overprovisioning, and vice versa.

V. Threats to Validity

We identify some external threats that could impact the generalizability of our findings. One such threat stems from the use of only one microservice benchmark application for the evaluation, potentially limiting the application of our findings to other microservice-based systems. However, given the widespread use, heterogeneity, and size of the selected benchmark application, we believe that our results can be applicable to other microservice applications or real-world settings. Additionally, the initial resource configurations of microservices, such as resource request and limit values, play a crucial role in the claimed improvement margin in our study. While we observed improvements in resource utilization using the benchmark application with default resource configurations, variations in initial resource configurations could result in smaller or larger improvements. Another external threat emerges from the comparison of our study with the Kubernetes baseline HPA to substantiate our findings, instead of multiple available alternative HPAs. However, the widespread adoption of Kubernetes in both industrial and academic settings mitigates this concern, providing a solid foundation for validating our study. Lastly, a potential threat lies in the variation of Smart HPA behavior under different workload profiles. The selected workload profile follows a pattern of increasing and sustained high workloads, crucial for creating resource-constrained scenarios for HPAs. Therefore, Smart HPA has the potential for consistent performance across a range of workload profiles, including those capable of inducing resource-constrained scenarios.

VI. Conclusion and Future Work

We introduce Smart HPA, featuring a hierarchical architecture that integrates both centralized and decentralized architectural styles to perform horizontal auto-scaling in microservices. Within this hierarchical architecture, decentralized managers monitor microservice resource metrics, like CPU usage, while the centralized manager is selectively activated in resource-constrained environments to efficiently transfer resources between microservices, minimizing communication overhead. In our experiments, Smart HPA outperforms Kubernetes HPA with default configurations for a benchmark application. It reduces resource overutilization by 5x, overutilization time by 2x, and overprovisioning by 7x. Additionally, it eliminates underprovisioning, improves resource allocation by 1.8x, and extends overprovisioning time by 10x.

We have identified several areas for improvement that we plan to address in our future work. One such avenue involves employing AI-based predictive methods, such as time series analysis of workload and resource utilization. This enables Smart HPA to operate with both proactive and reactive auto-scaling mechanisms. Moreover, evaluating Smart HPA with alternative scaling policies, such as queuing theory-based approaches, and assessing metrics like response time and communication overhead will strengthen and validate its flexibility. Furthermore, during our experiments, we observed the startup time of microservice containers significantly impacts the efficiency of auto-scaling operations. As such, a promising avenue for future research involves reducing startup time to expedite auto-scaling in microservice architectures.

References

[1] E. Pimentel, W. Pereira, P. H. M. Maia, M. I. Cortés, *et al.*, "Self-adaptive microservice-based systems-landscape and research opportunities," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 167–178, IEEE, 2021.

[2] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Dynamic multi-metric thresholds for scaling applications using reinforcement learning," *IEEE Transactions on Cloud Computing*, 2022.

[3] H. Ahmad, I. Dharmadasa, F. Ullah, and M. A. Babar, "A review on c3i systems' security: Vulnerabilities, attacks, and countermeasures," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–38, 2023.

[4] A. Gördén, "Predicting resource usage on a kubernetes platform using machine learning methods," 2023.

[5] J. Dobies and J. Wood, *Kubernetes operators: Automating the container orchestration platform*. O'Reilly Media, 2020.

[6] F. Soppelsa and C. Kaewkasi, *Native docker clustering with swarm*. Packt Publishing Ltd, 2016.

[7] G. Dumpleton, *Deploying to OpenShift: a guide for busy developers*. " O'Reilly Media, Inc.", 2018.

[8] N. Zhou, H. Zhou, and D. Hoppe, "Containerization for high performance computing systems: Survey and prospects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2722–2740, 2022.

[9] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *IEEE International Conference on Autonomic Computing and Self-organizing Systems (ACSOS)*, pp. 28–37, IEEE, 2020.

[10] J. Liu, S. Zhang, Q. Wang, and J. Wei, "Coordinating fast concurrency adapting with autoscaling for slo-oriented web applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3349–3362, 2022.

[11] M. ZargarAzad and M. Ashtiani, "An auto-scaling approach for microservices in cloud computing environments," 2023.

[12] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *1st ACM Symposium on Cloud Computing (SoCC)*, pp. 241–252, 2010.

[13] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020.

[14] A. Abdel Khaleq and I. Ra, "Intelligent microservices autoscaling module using reinforcement learning," *Cluster Computing*, pp. 1–12, 2023.

[15] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *ACM Symposium on Cloud Computing (SoCC)*, pp. 427–441, 2021.

[16] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, 2020.

[17] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1994–2004, IEEE, 2019.

[18] E. D. Nitto, L. Florio, and D. A. Tamburri, "Autonomic decentralized microservices: The gru approach and its evaluation," *Microservices: Science and Engineering*, pp. 209–248, 2020.

[19] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[20] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Computing and Applications*, vol. 32, pp. 9745–9760, 2020.

[21] F. Rossi, V. Cardellini, and F. L. Presti, "Self-adaptive threshold-based policy for microservices elasticity," in *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, IEEE, 2020.

[22] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, pp. 76–107, Springer, 2013.

[23] Replication Package for SmartHPA. https://github.com/HussainAhmad05/Smart_HPA.git.

[24] M. R. Hossen and M. A. Islam, "A lightweight workload-aware microservices autoscaling with qos assurance," *arXiv e-prints, pp. arXiv–2202*, 2022.

[25] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pp. 1–5, IEEE, 2020.

[26] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *IEEE 10th international conference on cloud computing (CLOUD)*, pp. 472–479, IEEE, 2017.

[27] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "gym-hpa: Efficient auto-scaling via reinforcement learning for complex microservice-based applications in kubernetes," in *NOMS IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9, IEEE, 2023.

[28] B. Liu, R. Buyya, and A. Nadjaran Toosi, "A fuzzy-based auto-scaler for web applications in cloud computing environments," in *16th International Conference on Service-Oriented Computing (ICSOC)*, pp. 797–811, Springer, 2018.

[29] V. Persico, D. Grimaldi, A. Pescape, A. Salvi, and S. Santini, "A fuzzy approach based on heterogeneous metrics for scaling out public clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2117–2130, 2017.

[30] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pp. 64–73, IEEE, 2017.

[31] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[32] Z. Ding and Q. Huang, "Copa: A combined autoscaling method for kubernetes," in *IEEE International Conference on Web Services (ICWS)*, pp. 416–425, IEEE, 2021.

[33] J. Tong, M. Wei, M. Pan, and Y. Yu, "A holistic auto-scaling algorithm for multi-service applications based on balanced queuing network," in *IEEE International Conference on Web Services (ICWS)*, pp. 531–540, IEEE, 2021.

[34] P. Kang and P. Lama, "Robust resource scaling of containerized microservices with probabilistic machine learning," in *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 122–131, IEEE, 2020.

[35] N. S. Joshi, R. Raghuwanshi, Y. M. Agarwal, B. Annappa, and D. Sachin, "Arima-pid: container auto scaling based on predictive analysis and control theory," *Multimedia Tools and Applications*, pp. 1–18, 2023.

[36] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 217–228, 2016.

[37] L. Baresi and G. Quattrocchi, "A simulation-based comparison between industrial autoscaling solutions and cocos for cloud applications," in *IEEE International Conference on Web Services (ICWS)*, pp. 94–101, IEEE, 2020.

[38] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958–972, 2021.

[39] J. Yang, C. Liu, Y. Shang, B. Cheng, Z. Mao, C. Liu, L. Niu, and J. Chen, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Information Systems Frontiers*, vol. 16, pp. 7–18, 2014.

[40] 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, 2012.

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

[42] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning," in *IEEE international conference on web services (ICWS)*, pp. 489–497, IEEE, 2020.

[43] S. M. R. Nouri, H. Li, S. Venugopal, W. Guo, M. He, and W. Tian, "Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications," *Future Generation Computer Systems*, vol. 94, pp. 765–780, 2019.

[44] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with sla objective using q-learning," in *IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 85–92, IEEE, 2018.

[45] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning,” in *IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, IEEE, 2019.

[46] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, “Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows,” in *IEEE 39th international conference on distributed computing systems (ICDCS)*, pp. 122–132, IEEE, 2019.

[47] P. Arcaini, E. Riccobene, and P. Scandurra, “Modeling and analyzing mape-k feedback loops for self-adaptation,” in *IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 13–23, IEEE, 2015.

[48] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, “Delivering elastic containerized cloud applications to enable devops,” in *IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 65–75, IEEE, 2017.

[49] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, “Elascale: autoscaling and monitoring as a service,” *arXiv preprint arXiv:1711.03204*, 2017.

[50] Amazon Web Services. [Online]. https://www.amazon.com (Accessed: August 2, 2023).

[51] Amazon Elastic Kubernetes Service. [Online]. https://aws.amazon.com/eks (Accessed: August 10, 2023).

[52] Online Boutique. [Online]. https://github.com/GoogleCloudPlatform/microservices-demo.git (Accessed: August 23, 2023).

[53] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, “Benchmark requirements for microservices architecture research,” in *IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pp. 8–13, IEEE, 2017.

[54] B. Choi, J. Park, C. Lee, and D. Han, “phpa: A proactive autoscaling framework for microservice chain,” in *5th Asia-Pacific Workshop on Networking (APNet)*, pp. 65–71, 2021.

[55] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, “Automated testing and resilience of microservice’s network-link using istio service mesh,” in *31st Conference of Open Innovations Association (FRUCT)*, pp. 79–88, IEEE, 2022.

[56] Locust - An open source load testing tool. [Online]. https://locust.io (Accessed: August 28, 2023).