The Journal of Systems and Software 225 (2025) 112390

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Towards resource-efficient reactive and proactive auto-scaling for microservice architectures☆

Hussain Ahmad [a],*, Christoph Treude [b], Markus Wagner [c], Claudia Szabo [a]

[a] School of Computer and Mathematical Sciences, The University of Adelaide, Australia
[b] School of Computing and Information Systems, Singapore Management University, Singapore
[c] Department of Data Science and AI, Faculty of IT, Monash University, Australia

ARTICLE INFO

Keywords:
Software architecture
Auto-scaling
Microservices
Resource management
Self-adaptation
Kubernetes

ABSTRACT

Microservice architectures have become increasingly popular in both academia and industry, providing enhanced agility, elasticity, and maintainability in software development and deployment. 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, resource mismanagement, and financial losses. Furthermore, the inherent delay in initializing and terminating microservice pods hinders HPAs from timely responding to workload fluctuations, further exacerbating these issues. To address these concerns, we propose Smart HPA and ProSmart HPA, reactive and proactive resource-efficient horizontal pod auto-scalers respectively. Smart HPA employs a reactive scaling policy that facilitates resource exchange among microservices, optimizing auto-scaling in resource-constrained environments. For ProSmart HPA, we develop a machine-learning-driven resource-efficient scaling policy that proactively manages resource demands to address delays caused by microservice pod startup and termination, while enabling preemptive resource sharing in resource-constrained environments. Our experimental results show that Smart HPA outperforms the Kubernetes baseline HPA, while ProSmart HPA exceeds both Smart HPA and Kubernetes HPA by reducing resource overutilization, overprovisioning, and underprovisioning, and increasing resource allocation to microservice applications.

1. Introduction

Microservice architectures have gained widespread popularity in recent years (Luo et al., 2024), with several prominent enterprises, including Netflix, eBay, and Amazon, and military systems transitioning from monolithic architectures to microservices to enhance their service quality (Marques et al., 2024; Ahmad et al., 2023). In the microservice software design paradigm, complex applications are decomposed into a set of autonomous, loosely coupled services that are domain-oriented and structured to ensure manageable complexity (Jayalath et al., 2024). Each microservice has a defined functionality and communicates with others through lightweight interfaces such as the HTTP resource API (Abdulsatar et al., 2024). Microservice architectures accelerate application delivery and improve reliability, as each microservice is designed, developed, and deployed independently (Bakshi, 2017). In general, microservices are deployed using software containers, with container orchestration platforms being widely adopted for the runtime management of microservices (Rossi et al., 2022). While various container orchestration platforms such as Kubernetes (Dobies and Wood, 2020), Docker Swarm (Soppelsa and Kaewkasi, 2016), and Red Hat OpenShift (Dumpleton, 2018), are available (Zhou et al., 2022b), Kubernetes is the most widely adopted platform in both academia and industry (Rossi et al., 2020a).

The increased adoption of microservice architectures brings a new set of challenges, such as managing fluctuating workloads (Zhou et al., 2022a; da Silva et al., 2021). For example, Amazon experienced service outages in 2018 due to its failure to manage workload fluctuations (Choi et al., 2021). Similarly, companies such as Microsoft and Zoom faced service disruptions during the COVID-19 pandemic due to sudden surges in workload (Choi et al., 2021). This increased

☆ Editor: Uwe Zdun.

* Corresponding author.
E-mail addresses: hussain.ahmad@adelaide.edu.au (H. Ahmad), ctreude@smu.edu.sg (C. Treude), markus.wagner@monash.edu (M. Wagner), claudia.szabo@adelaide.edu.au (C. Szabo).

https://doi.org/10.1016/j.jss.2025.112390
Received 28 July 2024; Received in revised form 26 December 2024; Accepted 14 February 2025
Available online 25 February 2025

load depletes the allocated resources for active microservices, resulting in longer response times and service unavailability. Auto-scaling is an essential approach to address such issues as it dynamically manages computing resources among microservices in response to workload fluctuations without human intervention, ensuring compliance with Service Level Agreements (SLAs) (da Silva et al., 2021; ZargarAzad and Ashtiani, 2023). A prominent solution for executing auto-scaling of microservice deployments is the Horizontal Pod Auto-scaler (HPA) (Nguyen et al., 2020). HPAs analyze resource utilization, workload fluctuations, and SLA-defined user requirements to adjust microservice replica counts in response to fluctuating workloads (Abdel Khaleq and Ra, 2023; Nguyen et al., 2020).

Existing HPAs exhibit certain limitations, including the inability to scale a microservice deployment beyond the predefined resource limits (Nguyen et al., 2020), delayed responses to workload fluctuations (Ju et al., 2021), and architectural vulnerabilities prone to failures (Rossi et al., 2020a). In a microservice application deployment, each microservice is allocated specific resources (i.e., maximum replica limit) (ZargarAzad and Ashtiani, 2023; Nguyen et al., 2020), 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 (Nguyen et al., 2020). This limitation, in turn, leads to service unavailability, performance degradation, and financial losses (Baarzi and Kesidis, 2021). At the same time, less busy microservices in the application have surplus resources, leading to resource wastage and additional operational costs (Yu et al., 2020). Furthermore, microservice pod startup and termination time hinders HPAs from timely responding to workload fluctuations, which results in increased resource over- or under-provisioning (Al Qassem et al., 2023; Ju et al., 2021). For example, the process of microservice instantiation, such as retrieving data files and system libraries from the file system, can take tens of seconds (Choi et al., 2021). According to Google Borg, the median container startup latency is 25 s (Verma et al., 2015), while researchers have observed a 90th percentile latency of 15 s for microservice pod startup with a state-of-the-art auto-scaler (Fu et al., 2020), which results in inconsistencies between workload demand and resource allocation (Marie-Magdelaine and Ahmed, 2020).

In addition, HPA control architectures are implemented in either a centralized or decentralized manner. In a centralized setup, a single HPA manages the scaling decisions for all microservices within an application (Gias et al., 2019). This approach simplifies decision-making by providing a unified and holistic view of the state of microservices. However, it also introduces several critical limitations. For example, the reliance on a single control point creates a single point of failure, making a microservice architecture vulnerable to disruptions if the central controller fails (Rossi et al., 2020c). Moreover, centralized HPAs introduce delayed response times to workload fluctuations, as all scaling decisions must pass through the central controller (Antons and Arlinghaus, 2021). Furthermore, in large-scale deployments, the central HPA becomes a computational bottleneck, overwhelmed by the high volume of scaling computations and decisions it must process (Weyns et al., 2013). To overcome these limitations, decentralized HPA control architectures have been proposed (Nitto et al., 2020). However, a significant challenge with decentralized HPAs is the lack of coordination in scaling decisions across different microservices (Rossi et al., 2020a). This lack of synchronization results in conflicting or suboptimal scaling actions, leading to inefficient auto-scaling operations. Recent improvements to decentralized solutions have been proposed through master–worker (Imdoukh et al., 2020) and hierarchical (Rossi et al., 2020b) architectures. However, these improvements have not effectively addressed the communication overhead challenges (Weyns et al., 2013). These challenges lead to the following research questions: *RQ1: How can an HPA architecture be designed to mitigate the limitations of centralized and decentralized architectural styles? RQ2: What HPA scaling policy maximizes the resource utilization in microservice architectures? RQ3: What HPA scaling policy addresses microservice pod startup and termination delays while maximizing resource utilization?*

To address these challenges, we propose two types of auto-scalers: reactive and proactive. Our reactive auto-scaler, Smart HPA, optimizes auto-scaling by facilitating resource exchange among microservices in resource-constrained environments. On the other hand, our proactive auto-scaler, ProSmart HPA, preemptively manages resource demands to reduce delays from pod startup and termination. Both auto-scalers incorporate a two-layered hierarchical architecture that combines centralized and decentralized architectural styles. Specifically, Smart HPA is based on the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge Base) control framework, whereas ProSmart HPA extends the MAPE-K framework by incorporating an Artificial Intelligence (AI) component, resulting in a MAPE-KI control framework. The intelligence component interfaces with the MAPE stages, providing AI-driven assistance to MAPE operations for facilitating proactive adaptation and increased responsiveness to workload fluctuations. The lower layer of the hierarchical architecture consists of decentralized microservice managers, each dedicated to a specific microservice, for handling the scaling operations for their respective microservices. Each microservice manager employs a local scaling policy to determine the desired number of replica pods for their microservices in response to fluctuating workload. Conversely, the higher-level centralized application resource manager activates only in resource-constrained scenarios. It employs a global scaling policy to facilitate resource exchange among microservices to enable resource-efficient scaling of microservices. Our proposed auto-scalers may reduce the communication overhead by controlling the interactions between their decentralized and centralized components, as the level of overhead is closely tied to the frequency of these interactions (Weyns et al., 2013). Additionally, the hierarchical architecture incorporates decentralized auto-scalers, which help overcome the limitations of centralized systems, such as susceptibility to single points of failure (Rossi et al., 2020a). However, the focus of this study is to maximize the resource utilization for microservices within their allocated resources through both reactive (Smart HPA) and proactive (ProSmart HPA) manners, aiming to minimize resource overutilization, underprovisioning, and overprovisioning to achieve improved performance. In summary, our contributions are as follows:

- We propose a hierarchical architecture for HPAs that integrates centralized and decentralized architectural styles to capitalize on their advantages while addressing limitations in managing scaling operations for microservice applications.
- We introduce Smart HPA, featuring a reactive resource-efficient scaling policy that transfers resources from overprovisioned to underprovisioned microservices, facilitating scaling operations in resource-constrained environments.
- We introduce ProSmart HPA, featuring a proactive resource-efficient scaling policy that predicts resource demands and capacities of microservices, enabling proactive scaling operations and resource optimization among microservices in resource-constrained environments.
- We present a comprehensive experimental analysis demonstrating the advantages of Smart HPA over Kubernetes HPA, and ProSmart HPA over both Smart HPA and Kubernetes HPA, using a real-world microservice 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. ProSmart HPA outperforms both Smart HPA and Kubernetes HPA under highly resource-constrained conditions, reducing overutilization by 25.40% and 25.73%, overprovisioning by 28.24% and 86.43%, and underprovisioning by 25.34% and 26.73%, while improving resource allocation by 2.17% and 20.75%, respectively.

This paper extends our previous work on Smart HPA (Ahmad et al., 2024b) with the following key distinctions: (i) We introduce ProSmart HPA, which offers resource-efficient proactive auto-scaling for microservice architectures; (ii) We extend the MAPE-K control framework to MAPE-KI by integrating an AI component that enables proactive auto-scaling; (iii) Within ProSmart HPA's hierarchical architecture, we distinctly separate the analyze and plan components, each with specialized functions; (iv) We formulate scaling policies for both the microservice managers and the application resource manager within ProSmart HPA, which allow for proactive redistribution of resources among microservices by anticipating their demands in resource-constrained environments; (v) We conduct a new set of experiments with variations in the prediction window size; (vi) We perform a comparative performance analysis of ProSmart HPA with our previous work Smart HPA and Kubernetes HPA; and (vii) We discuss the implications of proactive auto-scaling of microservices for researchers and practitioners working in this field. Moreover, we provide a replication package for ProSmart HPA (Ahmad et al., 2024), which includes scripts and data required for reproducing, validating, and extending the ProSmart HPA.

The remainder of the paper is organized as follows: Section 2 reviews the background and related work on reactive and proactive auto-scaling in microservice architectures. Section 3 covers the architectural design and resource-efficient scaling policy of Smart HPA, while Section 4 discusses the proactive auto-scaler, ProSmart HPA. Section 5 describes the experimental setup and presents the evaluation results for both Smart HPA and ProSmart HPA. Section 6 analyzes the findings and discusses their implications for researchers and practitioners. Section 7 addresses threats to the validity of the study. Finally, Section 8 concludes the paper.

2. Background and related work

This section provides an overview of the existing scaling policies and control architectures used for HPAs and outlines the distinctive features of the proposed auto-scalers.

2.1. Scaling policies

Resource management in auto-scaling relies on scaling policies, which dictate how an HPA determines the number of replicas for a microservice deployment, considering workload variation, resource usage, and SLA requirements. In the existing literature, we have identified the following five distinct categories of scaling policies.

(i) Threshold-based scaling policy. Threshold-based policies are widely used in both academia (e.g., Hossen and Islam (2022), Balla et al. (2020), Nitto et al. (2020) and Al-Dhuraibi et al. (2017)) 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 (Santos et al., 2023). 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 (Yu et al., 2020). Furthermore, defining threshold-based policies for complex microservice infrastructures with resource contention and inter-dependencies is challenging (Rossi et al., 2020b).

(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 (Liu et al., 2018; Persico et al., 2017; Arabnejad et al., 2017). 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 (Qu et al., 2018). However, fuzzy-based policies also demand a comprehensive understanding of the application for defining fuzzy rules (Rossi et al., 2020b; Yu et al., 2020).

(iii) Queuing theory-based scaling policy. The existing literature (e.g., Ding and Huang (2021), Gias et al. (2019), Rossi et al. (2020a) and Tong et al. (2021)) 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 (Rossi et al., 2022). However, the accuracy of queuing models can be compromised when there are significant deviations from the exponential distribution in inter-arrival or service times (Rossi et al., 2020a; Kang and Lama, 2020). Also, queuing models provide approximate estimations, and fine-tuning their parameters requires thorough application profiling, which can be costly and time-consuming (Rossi et al., 2020b).

(iv) Control theory-based scaling policy. A control theory-based scaling policy (Joshi et al., 2023; Baresi et al., 2016; Baresi and Quattrocchi, 2020; Baarzi and Kesidis, 2021) fine-tunes system behavior by evaluating the current value of a scaling metric against its reference value within a feedback loop. For example, Baarzi and Kesidis (2021) 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 (Rossi et al., 2022).

(v) Artificial Intelligence-based scaling policy. An Artificial Intelligence (AI) based scaling policy utilizes Machine Learning (ML) and Reinforcement Learning (RL) models to estimate and forecast scaling metrics, enabling adaptive and efficient auto-scaling of microservices in both reactive and proactive manners (Toka et al., 2021). For regression-based auto-scalers (Yang et al., 2014; Islam et al., 2012; Roy et al., 2011), 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 (Rossi et al., 2022). For RL-based auto-scalers, RL agents determine scaling actions by engaging in a sequence of interactions with their environment (Santos et al., 2023). Model-free RL algorithms, such as Q-learning and SARSA (Zhang et al., 2020; Nouri et al., 2019; Horovitz and Arian, 2018), suffer from slow learning rates. This results in time-consuming auto-scaling of microservices (Yu et al., 2020; Rossi et al., 2020a). While model-based RL approaches (Rossi et al., 2019, 2020b; Yang et al., 2019) can address the slow convergence issue of model-free methods, they face issues related to scalability when applied in large-scale microservice architectures (Rossi et al., 2022). To mitigate the challenges of increased resource overprovisioning or underprovisioning caused by microservice pod startup and termination latency, HPAs incorporate proactive scaling mechanisms. These mechanisms include statistical models such as ARIMA (Rossi et al., 2020a) and SARIMA (Subramaniam and Subramaniam, 2023), as well as AI-based approaches (Al Qassem et al., 2023; Subramaniam and Subramaniam, 2023; Nguyen et al., 2022; Qiu et al., 2020), to forecast future resource demands based on historical data behavior for microservice auto-scaling. However, these proactive methods typically forecast demand within microservice resource capacities. As a result, they prove less effective in resource-constrained environments where microservices often require more resources than initially allocated.

2.2. 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 (Arcaini et al., 2015). In the existing literature, we have identified two distinct architectural styles for auto-scalers: (i) centralized, and (ii) decentralized.

(i) Centralized architecture. The majority of the existing HPAs, such as Gias et al. (2019), Barna et al. (2017) and Khazaei et al. (2017), 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 (Rossi et al., 2020a).

(ii) Decentralized architecture. To address the limitations of centralized architectures, in decentralized architectural style (Nitto et al., 2020; Nouri et al., 2019), 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 (Nguyen et al., 2020). 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 (Rossi et al., 2020a). This situation degrades the performance of microservice applications. To address this issue, recent works have proposed hierarchical (Rossi et al., 2020b,a) and master–worker (Rossi et al., 2020c; Imdoukh et al., 2020; Rossi et al., 2019) 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 (Weyns et al., 2013).

To summarize, the existing centralized and decentralized HPA architectures provide certain advantages, but they also face critical limitations, such as increased response times and bottlenecks, which hinder efficient microservice auto-scaling operations. In addition, existing auto-scaling policies, both reactive and proactive, have significant limitations. Reactive policies perform scaling operations in response to workload fluctuations. However, due to delays caused by pod initialization and termination times, as well as scaling decisions confined to pre-allocated resource capacities, these policies often result in inefficiencies such as increased resource overutilization and underprovisioning. On the other hand, proactive scaling policies attempt to forecast demand to address microservice pod initialization and termination delays, but they also typically make scaling decisions within the constraints of allocated resource capacities. Consequently, they are less effective in resource-constrained environments where microservices often require more resources than initially provisioned. This underscores the need for an HPA architecture that integrates the strengths of centralized and decentralized approaches while addressing their respective limitations. Such an architecture should also incorporate a proactive scaling policy where microservice scaling decisions are not constrained by their respective allocated resource capacities. Instead, it should enable dynamic resource sharing among microservices, as demonstrated in our previous work on Smart HPA (Ahmad et al., 2024b), while also introducing proactive resource exchange alongside forecasting microservice resource demand to ensure efficient proactive auto-scaling operations. The following section elaborates on Smart HPA, and the subsequent section (i.e., Section 4) introduces the proposed extension, ProSmart HPA.

3. Smart HPA architecture

In this section, we describe the hierarchical architecture and resource-efficient scaling policy 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 *Application Resource Manager*. To adapt to frequent resource congestion, Smart HPA incorporates the components of the MAPE-K framework (Arcaini et al., 2015) into both the decentralized Microservice Managers and the centralized Application Resource Manager for executing the auto-scaling of a microservice application. As presented in Fig. 1, a Knowledge Base, part of the MAPE-K framework, connects the Microservice Managers and the Application Resource Manager, facilitating the exchange of scaling-related data. In addition, this Knowledge Base provides critical auto-scaling status updates to developers and operators, promoting transparency and operational insight. The bottom half of Fig. 1 depicts the deployment of microservices on a Kubernetes Cluster, comprising a control plane and multiple worker nodes. Microservice pods run on these worker nodes, and Smart HPA dynamically manages their lifecycle by creating and removing pods in response to workload fluctuations. As shown by the red arrow in Fig. 1, scaling decisions are communicated to the Kubernetes control plane, which instructs kubelets on the worker nodes to adjust the number of pod replicas accordingly. Auto-scaling operations are informed by metrics collected from the Metrics Server and the user-defined SLAs.

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 Application Resource Manager. This manager employs a resource-efficient scaling policy, 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. 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. As a prototype implementation, we show the benefits of Smart HPA instantiated with a threshold-based scaling policy due to its straightforward implementation.

3.1. Microservice manager

The Microservice Manager is composed of the components of the MAPE-K framework (Arcaini et al., 2015). 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

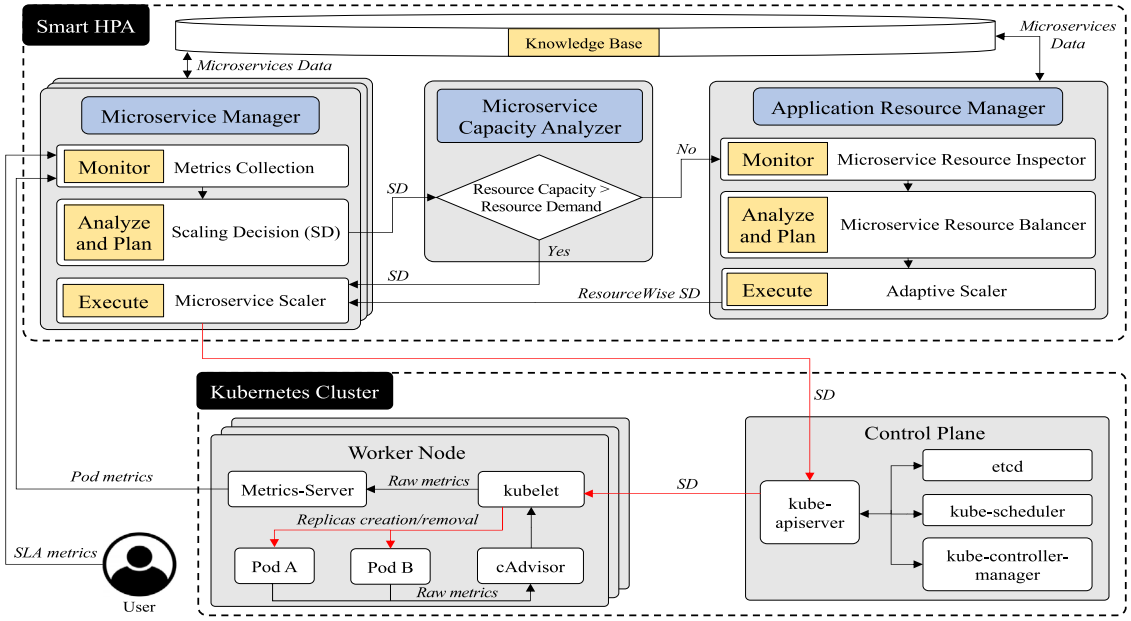


Fig. 1. Proposed hierarchical architecture of smart HPA.

Execute component of a Microservice Manager scales a microservice based on instructions from either the Microservice Capacity Analyzer or the Application Resource Manager (Fig. 1). Algorithm 3 outlines the functionality of a Microservice Manager dedicated to a microservice *i*.

3.1.1. 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.

3.1.2. 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 3 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 (Nguyen et al., 2020), within the Analyze and Plan components of all Microservice Managers to determine desired replica counts for their respective microservices. In Algorithm 3, 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 3 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 3 outline the scaling decisions *SD* corresponding to different types of detected violations.

Algorithm 2 Application Resource Manager

M: total number of microservices, *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: $ResSD_i, ResDR_i$ $\forall i=1,\cdots,M$

function Microservice Resource Inspector

// Finding residual and required resources for all services
1: $Overprov$ =[ ], $Underprov$ =[ ]
2: **for** $i = 1$ to M **do**
3: **if** $DR_i > maxR_i$ **then**
4: $RequiredR_i = DR_i - maxR_i$
5: $RequiredRes_i = RequiredR_i \times ResReq_i$
6: $Underprov$.append($RequiredRes_i$)
7: **else**
8: $ResidualR_i = maxR_i - DR_i$
9: $ResidualRes_i = ResidualR_i \times ResReq_i$
10: $Overprov$.append($ResidualRes_i$)
11: **end if**
12: **end for**
13: **return** $Underprov, Overprov$

function Microservice Resource Balancer

// Resource transfer from overprovision to underprovision services; suggesting feasible desired replicas (*FeasibleR*) and resource capacities (*UmaxR*) for all services
14: $FeasibleR$ =[ ], $UmaxR$ = [ ]
15: U = length($Underprov$), O = length($Overprov$)
16: $TotalOverprov = sum(Overprov)$ // total residual resource
// Resource reallocation for underprovisioned microservices
17: $Underprov \leftarrow$ Dsort($Underprov$) // sort in descending order
18: **for** $i = 1$ to U **do** // U = total underprovision microservices
19: $TotalR_i = TotalOverprov / ResReq_i$
20: **if** $TotalR_i \geq RequiredR_i$ **then**
21: $FeasibleR_i, UmaxR_i = DR_i$
22: **else if** $TotalR_i \in [1, RequiredR_i)$ **then**
23: $FeasibleR_i, UmaxR_i = \text{floor}(TotalR_i) + maxR_i$
24: **else**
25: $FeasibleR_i, UmaxR_i = maxR_i$
26: **end if**
27: $UsedRes_i = (FeasibleR_i - maxR_i) \times ResReq_i$
28: $TotalOverprov = TotalOverprov - UsedRes_i$
29: **end for**
// Resource reallocation for overprovisioned microservices
30: $Overprov \leftarrow$ Asort($Overprov$) // sort in ascending order
31: **for** $i = 1$ to O **do** // O = total overprovisioned microservices
32: $TotalR_i = TotalOverprov / ResReq_i$
33: **if** $TotalR_i \geq ResidualR_i$ **then**
34: $UmaxR_i = maxR_i$
35: **else if** $TotalR_i \in [1, ResidualR_i)$ **then**
36: $UmaxR_i = \text{floor}(TotalR_i) + DR_i$
37: **else**
38: $UmaxR_i = DR_i$
39: **end if**
40: $FeasibleR_i = DR_i$
41: $UsedRes_i = (maxR_i - UmaxR_i) \times ResReq_i$
42: $TotalOverprov = TotalOverprov - UsedRes_i$
43: **end for**
44: **return** $FeasibleR, UmaxR$

function Adaptive Scaler

// Updating desired replica counts, scaling decisions, and resource capacities for all microservices
45: **for** $i = 1$ to M **do**
46: **if** $FeasibleR_i == DR_i$ **then**
47: $ResSD_i = SD_i$
48: **else if** $FeasibleR_i \in (maxR_i, DR_i)$ **then**
49: $ResSD_i$ = Scale up
50: **else**
51: $ResSD_i$ = No Scale
52: **end if**
53: $maxR_i = UmaxR_i$, $ResDR_i = FeasibleR_i$
54: **end for**
55: $KB \leftarrow Underprov, Overprov, ResSD_i, ResDR_i, maxR_i$
56: **return** $ResSD_i, ResDR_i$
end

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).

3.2. 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 (ZargarAzad and Ashtiani, 2023). 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., Application Resource Manager) to exchange resources among microservices for executing scaling decisions. This purposeful activation of the centralized Application Resource Manager allows Smart HPA to potentially reduce communication overhead between its centralized and decentralized components.

3.3. Application resource manager

The centralized component of our hierarchical Smart HPA, the Application 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 a resource-efficient scaling policy outlined in Algorithm 2 for the Application Resource Manager to enable resource-wise scaling of microservices. Fig. 1 presents the MAPE-K components of the Application Resource Manager, which are distributed across three key components: Microservice Resource Inspector, Microservice Resource Balancer, and Adaptive Scaler.

3.3.1. Microservice resource inspector

Lines 1–13 of our scaling policy 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 3–6) and residual resources *Overprov* for overprovisioned microservices (lines 8–10).

3.3.2. 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, it 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 14–44 of Algorithm 2.

To prioritize addressing the needs of highly underprovisioned services (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 services' resource (*Underprov*) in descending order (line 17), and the overprovisioned services' resource (*Overprov*) in ascending order (line 30). 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 18–29). Consequently, the Microservice Resource Balancer reduces the resource capacities of overprovisioned microservices, indicating retrieval of residual resources from them (lines 31–43). 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 22–23). 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 24–25).

3.3.3. 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 45–56 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 55).

It is noteworthy that the proposed resource-efficient scaling policy (Algorithm 2) can integrate with other scaling policies (Section 2), and metrics, such as CPU usage and response time. This flexibility allows researchers and practitioners to select scaling policies and metrics according to specific requirements.

4. ProSmart HPA architecture

In this section, we describe the hierarchical architecture and resource-efficient scaling policy of ProSmart HPA. Fig. 2 presents the two-layered intelligent hierarchical architecture of ProSmart HPA, consisting of the Microservice Manager and the Application Resource Manager. We extend the MAPE-K control framework by incorporating an Artificial Intelligence (AI) component, resulting in a MAPE-KI control framework, to execute proactive microservice auto-scaling through predicted resource demand, capacity, and optimization. Both Microservice and Application Resource Manager utilize the MAPE-KI framework to oversee the proactive auto-scaling of microservices. To enhance clarity, we distinguish the MAPE-KI components of both layers by subscript notation, with "M" representing Microservice Manager components and "A" denoting Application Resource Manager components (e.g., Monitor_M and Monitor_A). As shown in Fig. 2, each layer is equipped with its own Knowledge Base, which facilitates seamless data sharing among the MAPE components within the layer. The Knowledge Base serves as a repository for resource metrics, scaling decisions, and operational insights. Fig. 2 also highlights the communication pathways, depicted by red arrows, that connect the layers and extend to the external environment where the microservice application is deployed on a Kubernetes Cluster. The Microservice Manager collects SLA and pod metrics from users and the Kubernetes Cluster, respectively, to inform its decisions. Scaling decisions made by both layers are communicated to the Kubernetes control plane, which executes these actions by directing the kubelets on worker nodes to create or remove microservice pod replicas in response to workload changes. In the lower layer, each microservice is managed by its own Microservice Manager. This dedicated allocation enables customized adjustments to AI algorithms, scaling metrics, and policies, according to the specific needs of each microservice. Furthermore, all Microservice Managers function autonomously in a decentralized fashion, running concurrently to gather and analyze microservice data within resource-rich environments. This results in enhanced data monitoring and analysis, facilitating faster adaptation compared to a sequential centralized approach. In resource-constrained scenarios where a microservice's demand exceeds its capacity, the Microservice Manager activates the centralized Application Resource Manager. This selective activation reduces communication overhead between the centralized Application Resource Manager and the decentralized Microservice Managers. The Application Resource Manager employs a global scaling policy to coordinate Microservice Managers, predict resource capacities for all microservices, and make essential resource adjustments accordingly. As a prototype implementation, we use the PROPHET model (Prophet) to predict both resource demands and resource capacities of microservices, known for handling unavailable data, shifts in trends, and outliers effectively (Prophet).

4.1. Microservice manager

The MAPE-KI controller of a Microservice Manager is responsible for gathering and analyzing data from a microservice to predict its resource demand to formulate and execute a scaling decision accordingly. If the predicted resource demand exceeds the microservice's capacity, the Plan_M component activates the Application Resource Manager to facilitate resource-efficient proactive auto-scaling of microservices. Alternatively, it triggers the Execute_M component to implement the scaling decision. Algorithm 3 delineates the local scaling policy of a Microservice Manager responsible for a microservice *i*.

Monitor_M: Metrics Collection. The Monitor_M gathers crucial SLA and pod metrics related to a microservice. Pod metrics include metrics such as resource utilization (*RM*) and current replica count (*CR*), which reflect the current state of a microservice. SLA metrics represent the requirements of users, such as maximum/minimum replica counts

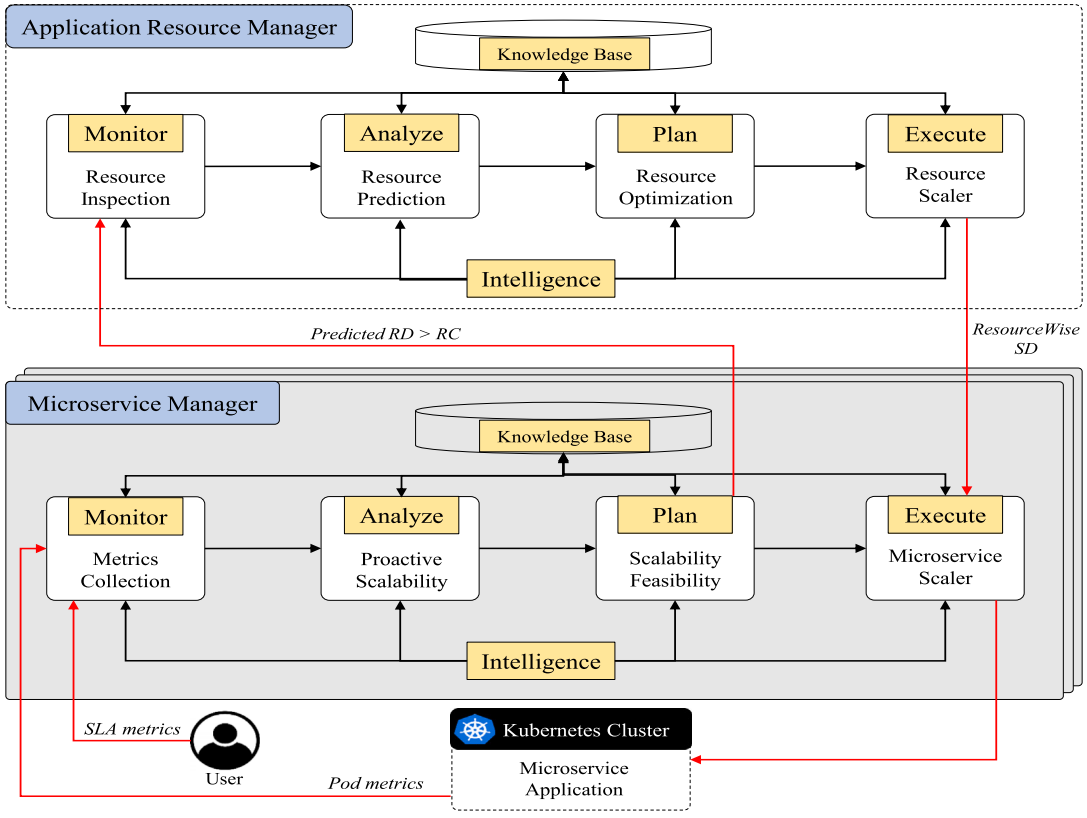


Fig. 2. Hierarchical MAPE-KI architecture of ProSmart HPA; RD: Resource Demand, RC: Resource Capacity, SD: Scaling Decision.

Algorithm 3 MICROSERVICE MANAGER

RM: resource metric value, *RMT:* resource metric threshold value, *ResReq:* resource request per replica, *CR:* current replica count, *PDR:* predicted desired replica count, *SD:* scaling decision, *minR:* minimum replica count, *maxR:* maximum replica count, *ARM:* Application Resource Manager, *KB:* knowledge base

Monitor: $RM_i, RMT_i, CR_i, ResReq_i, maxR_i, minR_i$

1: PDR_i = **PROPHET** (RM_i, RMT_i, CR_i)
2: **if** $PDR_i > CR_i$ **then**
3: SD_i = Scale up
4: **else if** $PDR_i < CR_i$ and $PDR_i \geq minR_i$ **then**
5: SD_i = Scale down
6: **else**
7: SD_i = No Scale
8: **end if**
9: **if** $PDR_i > maxR_i$ **then**
10: $PDR_i, SD_i, maxR_i$ = **ARM** $(PDR_i, SD_i, CR_i, maxR_i, ResReq_i)$
11: **end if**
12: $KB \leftarrow PDR_i, SD_i, CR_i, RM_i, RMT_i, ResReq_i, maxR_i, minR_i$
13: **Execute** (PDR_i, SD_i)

(*maxR, minR*), resource threshold value (*RMT*), and resource request value per replica (*ResReq*) for a microservice.

Analyze$_M$: Proactive Scalability. The Analyze$_M$ determines predicted resource demand, detects violations, and makes scaling decisions for its associated microservice. Algorithm 3 outlines this process in lines 1–8, summarized as follows: *Step 1: Resource Demand Prediction.* The Analyze$_M$ component employs the ML-based PROPHET[1] model (Prophet) to predict microservice resource demand (desired replica counts (PDR_i)). This prediction takes into account factors such as the current replica count (CR_i), resource utilization (RM_i), and resource threshold value (RMT_i) of the respective microservice (line 1). The development, training, and deployment process of the PROPHET model is explained in Section 5. *Step 2: Violation Detection.* When the predicted desired replica count (PDR_i) differs from the current replica count (CR_i), it results in a violation, indicating that the microservice's resource utilization is expected to deviate from its threshold limit in the future. Algorithm 3 outlines the criteria for detecting violations in lines 2 and 4, highlighting the necessity for microservice auto-scaling. *Step 3: Scaling Decision.* Detected violations trigger a scale up, scale down, or no scale decision (SD_i) for a microservice (lines 2–7).

Plan$_M$: Scalability Feasibility. The Plan$_M$ evaluates the feasibility of implementing the scaling decision formulated by the Analyze$_M$ for the corresponding microservice. If the predicted resource demand of a microservice falls within its capacity bounds (i.e., $PDR_i \leq maxR_i$), the Plan$_M$ directs the Execute$_M$ component to proceed with the scaling decision (line 13). Conversely, if the microservice necessitates more resources than its capacity permits (i.e., $PDR_i > maxR_i$), the

[1] ProSmart HPA offers flexibility with AI models and algorithms for executing proactive auto-scaling of microservices (Section 1), and PROPHET is used only as a prototype example.

Algorithm 4 Application Resource Manager

PmaxR: predicted maximum replica count, *AddedRes:* added resource, *RemovedRes:* removed resource, *TotalRes:* total resource of a microservice, *ResDR:* resource-wise desired replica count, *ResSD:* resource-wise scaling decision

Monitor: $PDR_i, SD_i, CR_i, maxR_i, ResReq_i$
1: $PmaxR_i =$ **PROPHET** $(PDR_i, maxR_i, CR_i)$
2: **if** $PmaxR_i > maxR_i$ **then**
3: $AddedRes = (PmaxR_i - maxR_i) \times ResReq_i$
4: **else**
5: $RemovedRes = (maxR_i - PmaxR_i) \times ResReq_i$
6: **end if**
7: **while** $AddedRes \neq RemovedRes$ **do**
8: **if** $AddedRes > RemovedRes$ **then**
9: $RemovedRes \leftarrow$ Extract resources from overprovisioned services
10: **else**
11: $AddedRes \leftarrow$ Add resources to underprovisioned services
12: **end if**
13: $PmaxR_i = TotalRes_i \div ResReq_i$
14: **end while**
15: **if** $PmaxR_i \geq PDR_i$ **then**
16: $ResSD_i = SD_i, ResDR_i = PDR_i$
17: **else if** $PmaxR_i \in (CR_i, PDR_i)$ **then**
18: $ResSD_i =$ Scale up, $ResDR_i = PmaxR_i$
19: **else**
20: $ResSD_i =$ No Scale, $ResDR_i = CR_i$
21: **end if**
22: $KB \leftarrow ResDR_i, ResSD_i, PmaxR_i$
23: **Execute** $(ResDR_i, ResSD_i, PmaxR_i)$

Plan_M triggers the central component of ProSmart HPA (the Application Resource Manager) to predict microservice resource capacity and recommend resource-wise predicted desired replica counts, scaling decision, and resource capacity for the microservice (lines 9–10). This deliberate activation of the Application Resource Manager enables ProSmart HPA to potentially minimize communication overhead between its decentralized and centralized components.

Execute_M: Microservice Scaler. The Execute_M implements the scaling decision (SD_i) for the respective microservice by creating the predicted desired replica counts (PDR_i) for that microservice (line 13).

Intelligence_M. The Intelligence_M component is pivotal for enhancing the efficiency of MAPE operations within the Microservice Manager (da Silva et al., 2021; Gheibi et al., 2021). For implementing the prototype of ProSmart HPA, we use the intelligence in the Analyze_M component, where the ML-based PROPHET model (Prophet) is utilized to predict the resource demand of microservices. This predictive capability preemptively reduces the microservice pod startup and termination time, thereby mitigating the underprovisioning and overprovisioning of resources.

4.2. Application resource manager

The Application Resource Manager serves as the centralized component and top layer of our hierarchical MAPE-KI architecture, as illustrated in Fig. 2. It predicts microservice resource capacities, fosters coordination among decentralized Microservice Managers to facilitate proactive resource distribution, and renders resource-wise scaling decisions based on the predicted resource capabilities in resource-constrained environments (i.e., $PDR_i > maxR_i$). Algorithm 4 delineates the global scaling policy for the Application Resource Manager, aiming to facilitate resource-efficient microservice scaling operations.

Monitor_A: Resource Inspector. The Monitor_A collects resource-related data from the Plan_M components of Microservice Managers for all microservices. This data encompasses predicted resource demand (PDR_i), scaling decisions (SD_i), current replica count (CR_i), resource capacity ($maxR_i$), and resource request value ($ResReq_i$) for each microservice in an application.

Analyze_A: Resource Capacity Prediction. The Application Resource Manager is activated in resource-constrained environments when the predicted resource demand of a microservice surpasses its resource capacity. To accommodate the predicted resource demands of microservices, we predict resource capacities for all microservices. We utilize the ML-based PROPHET model for predicting microservice resource capacities. The PROPHET model takes into account factors such as predicted resource demand (PDR_i), current replica count (CR_i), and current resource capacity ($maxR_i$) to predict the resource capacities ($PmaxR_i$) for microservices (Algorithm 4, Line 1). Section 5 details the development, training, and deployment of the PROPHET model.

Plan_A: Resource Optimization. Once the predicted resource capacities are determined, we compute the added resource ($AddedRes$) and removed resource ($RemovedRes$) for all microservices, following the steps outlined in lines 2–6 of Algorithm 4. This allows us to verify whether the resources added by PROPHET correspond to those removed, as any disparity could disrupt the overall resource capacity available for the entire microservice application. In scenarios where the added resource exceeds the removed resource (i.e., $AddedRes > RemovedRes$), we systematically remove resources, starting from the most overprovisioned microservice down to the least, prioritizing the resource requirements of highly underprovisioned microservices until we balance the added and removed resources (lines 8–9). Conversely, if the removed resource surpasses the added resource (i.e., $RemovedRes > AddedRes$), we incrementally add resources, starting with the most underprovisioned microservice and proceeding to the least, ensuring the overall resource capacity for the entire microservice application is maintained, until we achieve equilibrium between the added and removed resources (lines 10–11). Lastly, the Plan_A adjusts the resource capacities of microservices according to the addition or removal of resources from microservices (line 13).

Execute_A: Resource Scaler. The Execute_A determines the resource-efficient desired replica counts ($ResDR_i$) and makes scaling decisions ($ResSD_i$) accordingly for all microservices based on their predicted resource capacities ($PmaxR_i$), as outlined in lines 15–20 of Algorithm 4. If the predicted resource capacities are greater than or equal to the predicted resource demands ($PmaxR_i \geq PDR_i$), the resource-efficient scaling decision and desired replica counts remain the same as those determined by the Microservice Managers (lines 15–16). However, Execute_A calculates the maximum feasible desired replica count for microservices (lines 17–18) if the predicted resource capacities are insufficient to meet the resource demands. Furthermore, in highly resource-constrained scenarios where the predicted resource capacities of microservices equal their current replica counts, no scaling action is undertaken (lines 19–20). Finally, the $ResSD_i$, $ResDR_i$, and $PmaxR_i$ values for each microservice are transmitted to the respective Execute_M to implement the resource-wise scaling decision (Fig. 2).

Intelligence_A. The Intelligence_A component enhances the MAPE operations of the Application Resource Manager. We employ the ML-powered PROPHET model in the Analyze_A component to predict resource capacities of microservices for the prototype implementation of ProSmart HPA. This prediction enables microservices to align with their predicted resource demands.

It is noteworthy that both Microservice and Application Resource Managers store all monitored and processed data within the Knowledge Base of their MAPE-KI controllers (Algorithm 3, line 12 and Algorithm 4, line 22). The Knowledge Base functions as a centralized repository,

ensuring that all relevant information is readily available to support autonomous operations and provide valuable insights for stakeholders. By providing shared access, it enables MAPE components of both managers to retrieve and utilize data for their respective processes, such as analysis, planning, and execution. Moreover, the Knowledge Base enhances transparency and situational awareness, empowering key stakeholders to make informed decisions. For example, developers can analyze logs to understand resource allocation trends and refine auto-scaling operations. Practitioners can review logs to ensure that scaling decisions comply with organizational policies and verify that scaling actions do not violate SLAs.

5. Experimental evaluation

In this section, we evaluate the proposed hierarchical architecture and resource-efficient scaling policies of Smart HPA and ProSmart HPA against the Kubernetes baseline HPA through two broad phases: experimental setup and execution. In the experimental setup phase, we begin by configuring the experimental environment, setting up a Kubernetes cluster with a specified number of nodes and their resource capacities. We then select a benchmark microservice application to serve as the testbed for our evaluation. Following this, we configure the load testing environment, which includes selecting a suitable load testing tool, defining the load test profile, and setting the test duration. We identify evaluation metrics, focusing on CPU utilization as a critical measure of resource efficiency, and design nine experimental scenarios by varying resource thresholds and resource capacities to test the proposed auto-scalers under different conditions. For ProSmart HPA, we integrate the PROPHET ML-based prediction model to forecast resource demands and capacities of microservices to enable proactive auto-scaling. In the execution phase, the experiments are conducted by deploying the benchmark application on the Kubernetes cluster, applying the configured load testing settings, and running the defined experimental scenarios. The details of these phases are provided in the following sections. Section 5.1 outlines the experimental setup process. Section 5.2 focuses on experiment execution, presenting and analyzing the results of Smart HPA in comparison with Kubernetes HPA. Finally, Section 5.3 discusses the performance of ProSmart HPA, comparing it with both Smart HPA and Kubernetes HPA.

5.1. Experimental setup

This section outlines the experimental setup, detailing the steps and processes involved in preparing the environment for evaluating the proposed auto-scalers.

Experiment Environment. We use Amazon Web Services (AWS) (Amazon, 2024b) to assess the performance of our resource-efficient auto-scalers. 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, 4 GB 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) (Amazon, 2024a) 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 and ProSmart HPA are hosted on a local machine, featuring an Intel Corei7 2.60 GHz CPU and 16 GB RAM. They are connected to the benchmark application running on AWS EKS through an AWS command-line interface. To implement the distributed microservice managers of our proposed auto-scalers, we employ *Python's multiprocessing module* (Aziz et al., 2021), a powerful tool for parallelism that enables each microservice auto-scaling operations to run in an isolated process. Unlike Python's threading module, which is constrained by the Global Interpreter Lock (GIL) and cannot achieve true parallelism, the multiprocessing module creates separate processes with independent memory spaces (Aziz et al., 2021), thereby improving the efficiency of auto-scaling actions.

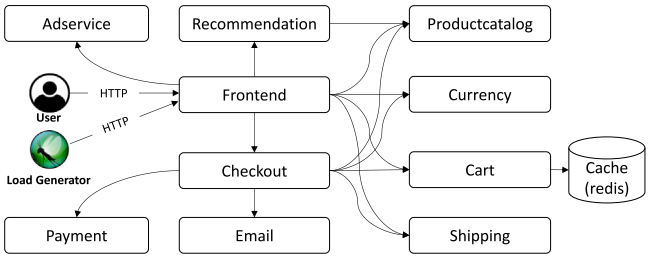
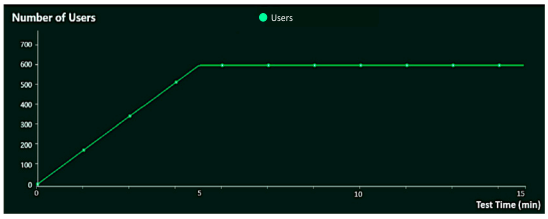


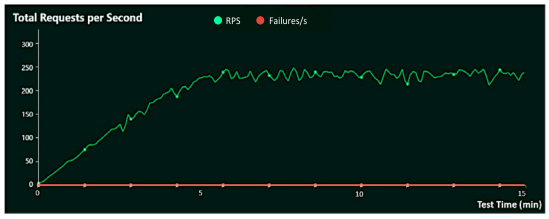
Fig. 3. Online boutique architecture (Google).

Benchmark Microservice Application. Smart HPA and ProSmart HPA can seamlessly integrate with any microservice application running on a Kubernetes cluster, highlighting their flexibility across diverse microservice applications. However, to evaluate these resource-efficient auto-scalers, we use a real-world microservice benchmark application, Online Boutique (Google), as it conforms to the benchmark selection criteria detailed in Aderaldo et al. (2017) and has been widely adopted by the research community, contributing to advancing the state-of-the-art in microservice architectures (Santos et al., 2023; Choi et al., 2021; Karn et al., 2022). Fig. 3 presents the architecture of the Online Boutique application. It 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 package also offers a load test script for scalability assessment (Google). We configure the benchmark application with its default resource settings: most microservices have a CPU request/limit of 100 m/200 m, *cart* and *adservice* have 200 m/300 m, and *redis* has 70 m/125 m. 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 for simulating end users to assess its scalability. The script simulates user interactions with the benchmark application, such as visiting the homepage, browsing products, adding items to the cart, viewing the cart, and checking out. To execute the load test script, we employ the (Locust) load testing tool. We run the Locust on the same local machine where our resource-efficient auto-scalers are deployed. As shown in Fig. 3, Locust sends HTTP requests to the benchmark application hosted on AWS EKS. Fig. 4 illustrates how Locust simulates users (Fig. 4(a)) and the associated workload (Fig. 4(b)) on the benchmark application. The load test is configured to run for a total duration of 15 min. In the initial 5 min, the test starts with 0 users and gradually increases, simulating the addition of 600 concurrent users with a 2-s spawn rate, which means that a new user is added every 2 s until the total reaches 600 users by the end of the 5-min ramp-up period. This initial phase serves to analyze the behavior of auto-scalers against increasing resource demand. Following this, there are 10 min of sustained high load, where all 600 concurrent users actively simulate HTTP requests. This sustained high load creates a resource-constrained scenario for both Smart HPA and ProSmart HPA. Due to the dynamic nature of the load test, with *wait_time = between(1, 5)* introducing randomized delays between task executions and a gradual ramp-up phase, the number of requests per user varies, but on average, each virtual user sends approximately 300 requests during the 15-min test.



(a) Simulated Users.



(b) Simulated Workload.

Fig. 4. Load test for benchmark application.

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 scaling policies of the proposed auto-scalers, 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 1. 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 1. It is important to note that Smart HPA and ProSmart HPA offer 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., Rossi et al. (2020a)) in Microservice Managers, the proposed auto-scalers can effectively carry out scaling operations with their resource-efficient scaling policies reported in Sections 3.3 and 4.2.

Resource Prediction. To execute the proactive auto-scaling, we use the PROPHET model (Prophet), developed by Facebook, to predict both resource demands and resource capacities of microservices. PROPHET is a time series forecasting model known for its robustness to unavailable data, shifts in trends, and outlier handling (Prophet). Given the 11 microservices in the benchmark application, we employ a total of 22 models. In particular, 11 PROPHET models are integrated into the Microservice Managers, each dedicated to predicting the resource demand of an individual microservice. Similarly, within the Application Resource Manager, another set of 11 PROPHET models is utilized, with each model assigned to predict the resource capacity of its respective microservice. Each model is trained separately using the time series dataset (Ahmad et al., 2024a) generated by its respective microservice while operating under Smart HPA. For instance, to predict resource demand, the dataset contains timestamps (ds) and corresponding resource demand (y), with additional regressors such as CPU usage, CPU threshold value, and current allocated resources to enhance the prediction accuracy of resource demand (y). We tune the hyperparameters of these models using a cross-validation approach. Fig. 5 shows the regression fit of training data against the PROPHET model's predictions for resource demand in a recommendation service, with uncertainty intervals at an 80% confidence level.

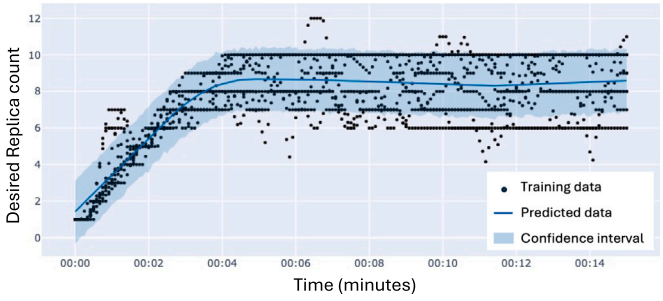


Fig. 5. PROPHET model fit.

Table 1
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 (min)	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 (min)	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 (min)	Total duration during which at least one microservice operates with insufficient CPU resource.

milliCPU: 1/1000th of a CPU core; *CPU Demand:* Total required CPU resources; *CPU Capacity:* Total allocated CPU resources.

Experimental Scenarios. As discussed in Section 1, the performance of HPAs is influenced by resource threshold configurations and resource capacities of microservices. Therefore, to determine the effectiveness of the proposed auto-scalers, 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%.

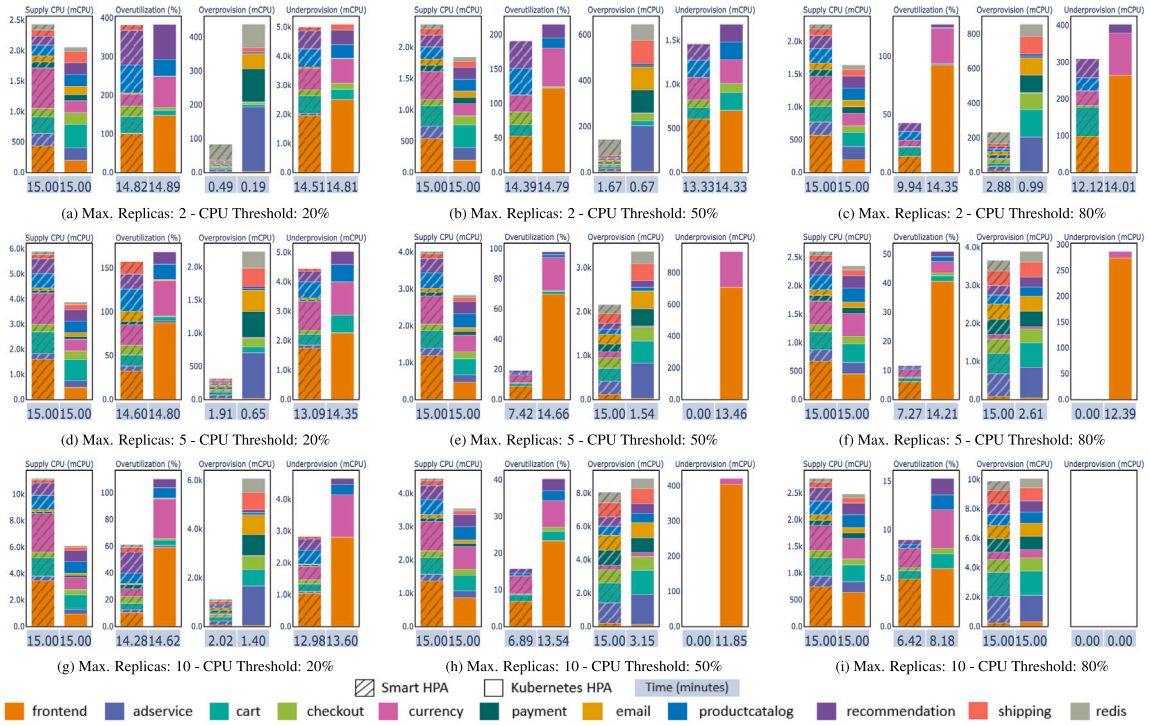


Fig. 6. Results for all experimental scenarios: Lower values, better performance (except Supply CPU and Overprovision Time). For a clearer view, refer to the detailed version of this figure in the online appendix[2].

5.2. Smart HPA: Results and discussion

In this section, we present and discuss the results of our load testing experiments conducted with the defined load testing configurations on the selected benchmark application, deployed on a Kubernetes cluster configured using the aforementioned experimental environment. The results are analyzed in terms of the identified evaluation metrics, with the experiments executed across the nine experimental scenarios.

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. In Fig. 6, 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. In the following, we discuss 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 scenario 5R-50% (Fig. 6(e)), Smart HPA shows no CPU underprovisioning, while Kubernetes records 934.04 m CPU underprovisioning for 13.46 min. This marks the highest improvement in both CPU Underprovision and Time across all nine scenarios, attributed to the efficient CPU resource balancing of Smart HPA. Consequently, Smart HPA results in 19.30% CPU Overutilization for 7.42 min, compared to Kubernetes HPA's 98.06% CPU Overutilization for 14.66 min, 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. 6(a)), 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 HPAs 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 when microservices experience no CPU underprovisioning. 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. 6(i)), 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. 6(g)), Smart HPA supplies 1.83x more CPU resources (11 188.76 m) to the benchmark application compared to the 6110.41 m Supply CPU provided by Kubernetes HPA. Conversely, in scenario 10R-80% (Fig. 6(i)), where no CPU underprovisioning occurs, Smart HPA holds a slight performance edge over Kubernetes HPA, supplying 1.11x more CPU resources (2771.42 m)

[2] https://doi.org/10.6084/m9.figshare.28028417.

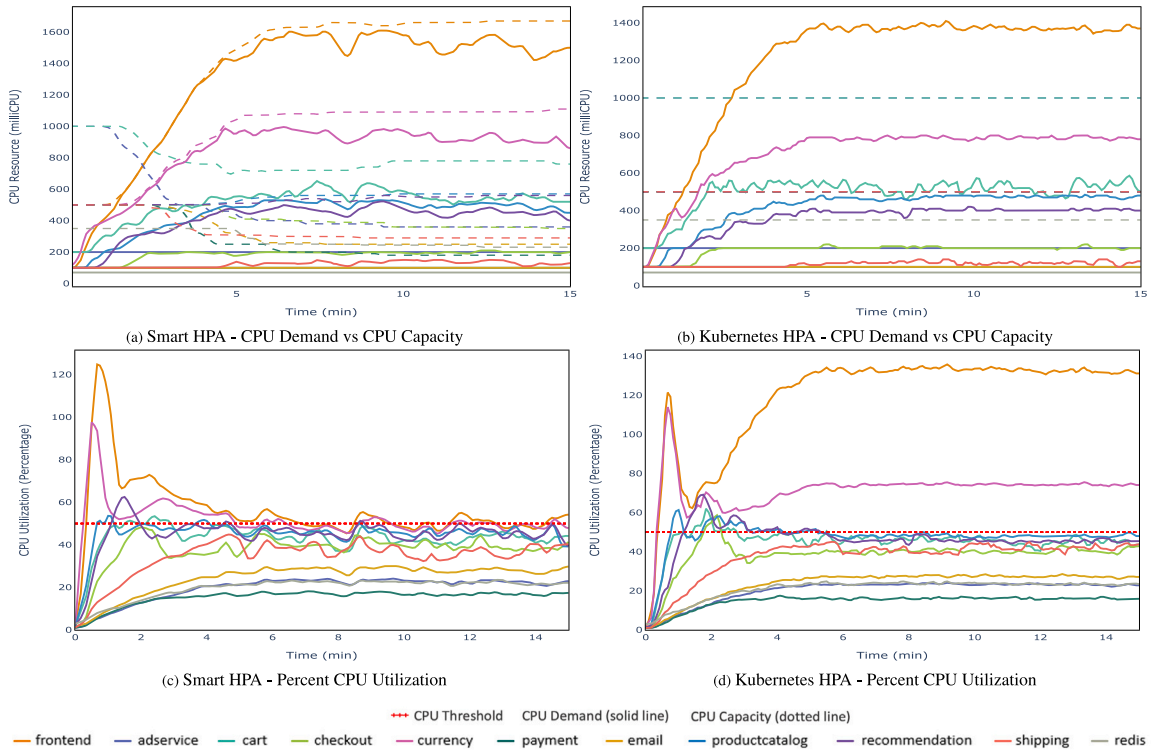


Fig. 7. Smart HPA vs. Kubernetes HPA: CPU utilization, CPU demand, and CPU capacity for the scenario 5R-50%.

compared to 2478.62 m CPU (Kubernetes HPA) to the benchmark application during the 15-min load test.

> Smart HPA shows optimal performance when overprovisioned microservices have a moderate surplus of resources (5R-50%). It performs comparably to Kubernetes HPA when overprovisioned microservices lack adequate resources (2R-20%).

Smart HPA vs. Kubernetes HPA: Adaptive Behavior Comparison. To understand the comparative behavior of Smart HPA and Kubernetes HPA during the load test, we present an entire scenario 5R-50% in Fig. 7. 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 Figs. 7(c) and 7(d). The rise in CPU utilization of microservices leads to an increase in their CPU demand, a pattern observed for both Smart HPA (Fig. 7(a)) and Kubernetes HPA (Fig. 7(b)).

In the case of Smart HPA, as shown in Fig. 7(a), around 1.50 min into the test, the CPU demand for the *frontend* exceeds its allocated CPU capacity of 500 m. At this point, the Microservice Capacity Analyzer triggers the Application Resource Manager. The manager identifies the *adservice* as the most overprovisioned, with 1000 m 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. 7(a)). Similarly, when the *currency* experiences a resource shortage around 2 min into the test, the Application 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 Application 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. 7(c), CPU utilization of both the *frontend* and *currency* experiences a decline, maintaining closer proximity to the 50% threshold.

In contrast, Kubernetes HPA does not facilitate resource sharing among microservices (Fig. 7(b)). This results in constant capacity levels (i.e., 500 m and 1000 m) 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. 7(d)). Therefore, when we compare Smart HPA to Kubernetes HPA (Fig. 6(e)), 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 934.04 m CPU Underprovision for 13.46 min.

> The resource exchange capability of Smart HPA 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. 6, 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 5887.53 m to 4013.82 m and further to 2601.18 m, respectively. This suggests that, as the CPU threshold increases, microservices operate effectively with fewer replicas. Furthermore, with increasing CPU threshold values, Fig. 6 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 occurs because each

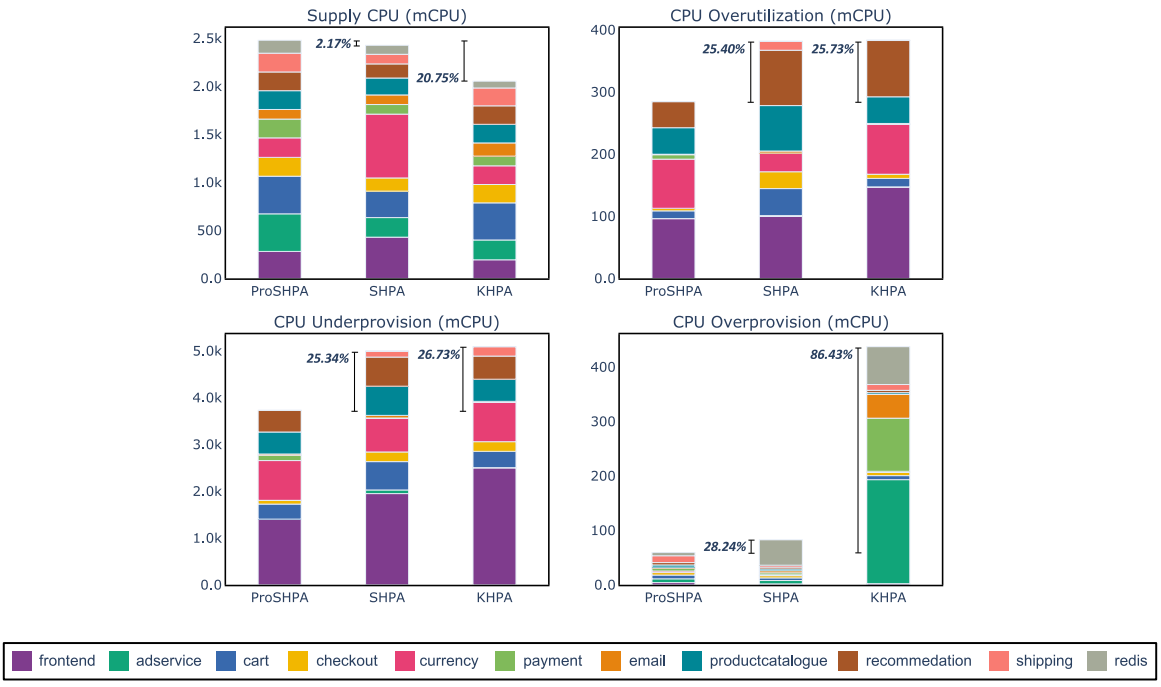


Fig. 8. Performance of ProSmart HPA (ProSHPA) vs. Smart HPA (SHPA) and Kubernetes HPA (KHPA): lower values preferred (except for Supply CPU).

microservice replica 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. 6 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 82.67 m, 315.40 m, and 1110.91 m (Fig. 6). 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. 6, scenario 10R-20% records the highest Supply CPU of 11 188.76 m for Smart HPA and 6110.41 m 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 (9864.39 m) and Overprovision Time (15 min) and the lowest CPU Overutilization (8.93%) and Overutilization Time (6.42 min), with zero CPU Underprovision and Time for Smart HPA.

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

5.3. ProSmart HPA: Results and discussion

We evaluate the performance of ProSmart HPA against both the Smart HPA and the Kubernetes HPA. As shown above, the most challenging scenario for Smart HPA is the 2R-20% scenario, where it shows minimal improvement over Kubernetes HPA due to limited additional resources, restricting its ability to redistribute resources to manage workload fluctuations. Therefore, we evaluate the performance of ProSmart HPA in this highly resource-constrained scenario to compare the effectiveness of proactive auto-scaling against reactive auto-scaling. In addition, to mitigate HPA response delays caused by microservice pod startup and termination, we explore the impact of changing the prediction window size for ProSmart HPA across six intervals: 5, 10, 15, 20, 25, and 30 s. These window sizes are selected based on the duration of a microservice pod startup and termination time, which are around 15 s and 30 s, respectively, in our experimental setup. This allows us to examine how changes in the prediction window size correlate with pod startup and termination latency.

We set the CPU threshold to 20% and the resource capacity to 2 maximum replicas for all microservices to create the scenario 2R-20%. We conduct load tests on the benchmark application deployed on AWS EKS, integrated with ProSmart HPA, using the prediction window sizes: 5, 10, 15, 20, 25, and 30 s. Similar to the experimental evaluation of Smart HPA, we run the load test 10 times for each prediction window configuration, averaging the results to ensure reliability.

Comparative Analysis: ProSmart HPA vs. Smart HPA and Kubernetes HPA. Fig. 8 presents a comparison of the optimal performance of ProSmart HPA, with a 25-s prediction window, against Smart HPA and Kubernetes HPA in terms of evaluation metrics. Each of the 11 microservices is color-coded to show microservice-level CPU resources, while the bar height reflects application-level CPU resources for all three auto-scalers. Additionally, the improvement margins of ProSmart HPA over Smart HPA and Kubernetes HPA are highlighted. In the highly resource-constrained environment of 2R-20%, we note a modest 2.17% improvement in Supply CPU with ProSmart HPA compared to Smart HPA. The marginal improvement is mainly due to the constrained resource availability in this environment. However, the proactive approach of ProSmart HPA to allocate the maximum

Fig. 9. ProSmart HPA Performance with changing prediction window size.

resources from this limited CPU resources results in a 2.17% improvement in Supply CPU. Similarly, compared to Smart HPA, ProSmart HPA shows the optimal reduction in CPU Overutilization and CPU Underprovision by 25.4% and 25.34%, respectively, due to the 25-s prediction window, which aligns closely with microservice pod startup time (15 s) and termination time (30 s). This microservice pod configuration activates microservices when predicted resource demands are imminent, thus reducing CPU Overutilization and CPU Underprovision. Moreover, we observe a 28.24% decrease in CPU overprovisioning compared to Smart HPA, demonstrating the efficient utilization of CPU resources by ProSmart HPA. In addition, ProSmart HPA significantly outperforms Kubernetes HPA, with improvements of 20.75%, 25.73%, 26.73%, and 86.43% for Supply CPU, CPU Overutilization, CPU Underprovision, and CPU Overprovision, respectively, due to its proactive auto-scaling and resource redistribution capabilities.

In summary, the intelligent hierarchical architecture, coupled with AI-driven scaling policies, enables ProSmart HPA to outperform the state-of-the-art Smart HPA and Kubernetes HPA, particularly under resource-constrained conditions.

ProSmart HPA Performance with Changing Prediction Window Size. ProSmart HPA consistently outperforms Smart HPA and Kubernetes HPA across all prediction window sizes: 5, 10, 15, 20, 25, and 30 s. Fig. 9 illustrates the fluctuations in the evaluation metric values for ProSmart HPA as the prediction window size changes. The improvement in evaluation metrics increases from 5 s to 25 s. At this point, ProSmart HPA achieves maximum improvement for all resource metrics, after which it begins to decrease. For example, CPU Overutilization decreases from 347.65 m at 5 s to 284.62 m at 25 s, indicating an increasing improvement from 8.88% to 25.40% compared to Smart HPA. Then, it rises again to 317.56 m at 30 s, with the improvement decreasing to 16.76% compared to Smart HPA. We observe only a marginal improvement in Supply CPU, from 2474.18 m to 2479.64 m, as the prediction window size changes from 5 s to 25 s, mainly due to the resource scarcity in the 2R-20% scenario. The fluctuation in the improvement of evaluation metrics is due to the set pod configurations that follow the default Kubernetes pod startup and termination times of 15 s and 30 s respectively. The improvement margins are relatively low before the 15-s prediction window size. The maximum improvement occurs between pod startup and termination time, specifically at 25 s. At 30 s, the improvement margins decrease again. This indicates that ProSmart HPA performs best when the prediction window size aligns closely with pod startup and termination times, facilitating timely resource allocation to microservices based on predicted demand and optimizing resource utilization and allocation.

Configuring the prediction window size closer to pod startup and termination times improves resource management in proactive microservice auto-scaling.

6. Discussion

Smart HPA introduces a reactive, resource-efficient scaling approach that dynamically reallocates resources from overprovisioned to underprovisioned microservices, enabling effective scaling in resource-constrained environments. Through a rigorous evaluation across nine scenarios, combining CPU thresholds (20%, 50%, 80%) with resource capacities capped at maximum replicas of 2, 5, and 10, Smart HPA consistently outperformed Kubernetes HPA. It excelled particularly when overprovisioned microservices maintained a moderate resource surplus (5R-50%), while performing on par with Kubernetes HPA under minimal surplus conditions (2R-20%). Moreover, the resource exchange capability of Smart HPA effectively prevents underprovisioning, yielding superior results across evaluation metrics. Furthermore, the experiments revealed a trade-off: increasing CPU thresholds or resource capacities reduces overutilization and underprovisioning but increases overprovisioning. These findings underscore the potential of Smart HPA as an effective solution for optimizing resource allocation and improving scalability in microservice architectures.

Building on this, ProSmart HPA integrates a proactive scaling policy that predicts resource demands and capacities, further enhancing efficiency in scaling operations under resource-constrained conditions. By addressing the limitations of Smart HPA, ProSmart HPA effectively manages challenges such as minimal resource availability, as demonstrated in the most constrained 2R-20% scenario. The experimental

results validate its superiority over both Smart HPA and Kubernetes HPA, leveraging its AI-driven predictive capabilities to optimize resource utilization. Fine-tuning the prediction window size to align with microservice pod startup and termination times further improves resource management, highlighting the importance of precise forecasting in proactive auto-scaling.

The empirical evaluation of both Smart HPA and ProSmart HPA offers valuable insights for both researchers and practitioners working with microservice architectures. For researchers, our study presents promising avenues for future research. One such direction involves investigating the integration of online machine learning techniques for workload and resource utilization, aiming to enhance proactive microservice auto-scaling for ProSmart HPA. Another important research direction involves extending the integration of the Intelligence component to other MAPE phases of ProSmart HPA. While our current ProSmart HPA prototype focuses on incorporating intelligence into the Analysis phase, extending this capability to the monitoring, planning, and execution phases could significantly improve its operational efficiency and adaptability in auto-scaling. Furthermore, another avenue we plan to explore involves conducting a comprehensive evaluation of both Smart HPA and ProSmart HPA across diverse workload profiles, such as dynamic user arrivals and departures, to simulate realistic scenarios. This includes examining their performance on user-oriented metrics like response time, bandwidth, and communication overhead. We also aim to investigate their behavior under diverse scaling policies, including queuing theory-based approaches, and evaluate their effectiveness across benchmark microservice applications of varying sizes. In addition, we intend to compare the performance of both Smart HPA and ProSmart HPA against current state-of-the-art HPA solutions to highlight their relative strengths and limitations. For practitioners, our study provides practical insights into both reactive and proactive auto-scaling for microservices. It offers guidance on configuring resources, such as setting resource thresholds and capacities, to achieve optimal CPU utilization. Additionally, it explains the performance variations of proactive auto-scalers when adjusting prediction window sizes. These insights help practitioners make informed decisions about microservice deployment configurations and prediction window settings for effective auto-scaling.

7. Threats to validity

We acknowledge the internal, external, and construct threats that could impact the validity of our study. In terms of internal validity, several factors could influence the reliability of our findings. The initial resource configurations of microservices, such as resource request and limit values, are crucial to the improvement margins claimed in our study. While we observed improvements in evaluation metrics using the benchmark application with default resource configurations, variations in initial resource configurations could result in smaller or larger improvements in evaluation metrics, potentially affecting the consistency of our results. In addition, variations in the default configurations of Kubernetes pods, including pod startup and termination times, could also affect the observed improvement margins in the evaluation metrics of ProSmart HPA. Furthermore, a potential internal threat lies in the variation of Smart HPA and ProSmart 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, our resource-efficient auto-scalers have the potential for consistent performance across a range of workload profiles, including those capable of inducing resource-constrained scenarios.

For external validity, we acknowledge the external threats that may affect the generalizability of our findings. One external 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. Another external threat emerges from comparing Smart HPA only with the Kubernetes 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. Furthermore, variations in AI models could affect the performance of ProSmart HPA. However, the selected AI model, PROPHET, excels at handling time series data, which indicates that the ProSmart HPA could perform well with other similar AI models. In terms of construct validity, the study assumes that default resource configurations for microservices and Kubernetes pods, as well as the selected workload profile, accurately represent real-world scenarios. However, these constructs may not fully capture the diversity of real-world configurations or workload patterns, such as bursty or fluctuating loads, potentially impacting the validity of the evaluation.

8. Conclusion

We introduce Smart HPA and ProSmart HPA, reactive and proactive horizontal pod auto-scalers, designed to tackle current challenges in HPAs, including delayed responses to workload fluctuations, limitations in scaling microservices beyond predefined resource limits, and vulnerabilities in HPA architectural designs. These auto-scalers feature a hierarchical architecture based on the MAPE-K control framework, integrating both centralized and decentralized architectures to harness their strengths and mitigate their weaknesses. In ProSmart HPA, we enhance the MAPE-K framework to MAPE-KI, incorporating an AI component to forecast and manage resource demands and capacities for proactive microservice auto-scaling. Within this hierarchical setup, decentralized microservice managers continuously handle auto-scaling operations for individual microservices. The centralized application resource manager activates selectively in resource-constrained environments to facilitate efficient resource allocation between microservices, thereby reducing communication overhead and potential bottleneck issues.

Our experimental findings show that Smart HPA significantly outperforms Kubernetes HPA when tested with a microservice benchmark application. Specifically, Smart HPA reduces resource overutilization by 5x, decreases the overutilization time by 2x, and lowers overprovisioning by 7x compared to Kubernetes HPA. Moreover, it completely eliminates underprovisioning, improves resource allocation by 1.8x, and extends the overprovision time by 10x. Meanwhile, ProSmart HPA exceeds both Smart HPA and Kubernetes HPA in highly resource-constrained conditions, reducing overutilization by 25.40% and 25.73%, decreasing overprovisioning by 28.24% and 86.43%, cutting underprovisioning by 25.34% and 26.73%, and improving resource allocation by 2.17% and 20.75%, respectively. Furthermore, our findings uncover potential avenues for future research, advancing the state-of-the-art in both reactive and proactive auto-scaling of microservices. Our research also offers practical insights for practitioners working with microservice architectures, aiming to improve the state-of-the-practice in this area.

CRediT authorship contribution statement

Hussain Ahmad: Writing – original draft, Validation, Software, Methodology, Data curation, Conceptualization. **Christoph Treude:** Writing – review & editing, Supervision. **Markus Wagner:** Writing – review & editing, Supervision. **Claudia Szabo:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

We acknowledge Dr. Diksha Goel, Research Scientist at CSIRO's Data61, Australia, for her valuable feedback, which has enhanced the quality of this work.

Data availability

We provide the replication package of Smart HPA (Ahmad et al., 2024a) and ProSmart HPA (Ahmad et al., 2024), encompassing all scripts and data essential for reproducing, validating, and extending the results outlined in the paper.

References

Abdel Khaleq, A., Ra, I., 2023. Intelligent microservices autoscaling module using reinforcement learning. Clust. Comput. 1–12. http://dx.doi.org/10.1007/s10586-023-03999-8.

Abdulsatar, M., Ahmad, H., Goel, D., Ullah, F., 2024. Towards deep learning enabled cybersecurity risk assessment for microservice architectures. http://dx.doi.org/10.48550/arXiv.2403.15169, arXiv preprint arXiv:2403.15169.

Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P., 2017. Benchmark requirements for microservices architecture research. In: IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering. ECASE, IEEE, pp. 8–13. http://dx.doi.org/10.1109/ECASE.2017.4.

Ahmad, H., Dharmadasa, I., Ullah, F., Babar, M.A., 2023. A review on c3i systems' security: Vulnerabilities, attacks, and countermeasures. ACM Comput. Surv. 55 (9), 1–38. http://dx.doi.org/10.1145/3558001.

Ahmad, H., Treude, C., Wagner, M., Szabo, C., 2024. Replication package for ProSmart hpa. https://github.com/HussainAhmad05/ProSmart_HPA.

Ahmad, H., Treude, C., Wagner, M., Szabo, C., 2024a. Replication Package of Smart HPA. Available at: https://doi.org/10.6084/m9.figshare.25393408.v1. (Accessed 20 March 2024).

Ahmad, H., Treude, C., Wagner, M., Szabo, C., 2024b. Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures. In: 2024 IEEE 21st International Conference on Software Architecture. ICSA, pp. 46–57. http://dx.doi.org/10.1109/ICSA59870.2024.00013.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P., 2017. Autonomic vertical elasticity of docker containers with elasticdocker. In: IEEE 10th International Conference on Cloud Computing. CLOUD, IEEE, pp. 472–479. http://dx.doi.org/10.1109/CLOUD.2017.67.

Al Qassem, L.M., Stouraitis, T., Damiani, E., Elfadel, I.A.M., 2023. Proactive random-forest autoscaler for microservice resource allocation. IEEE Access 11, 2570–2585. http://dx.doi.org/10.1109/ACCESS.2023.3234021.

Amazon, 2024a. Amazon elastic kubernetes service (EKS). https://aws.amazon.com/eks. (Accessed 10 March 2024).

Amazon, 2024b. Amazon web services (AWS). https://www.amazon.com. (Accessed 6 March 2024).

Antons, O., Arlinghaus, J.C., 2021. Adaptive self-learning distributed and centralized control approaches for smart factories. Procedia CIRP 104, 1577–1582. http://dx.doi.org/10.1016/j.procir.2021.11.266.

Arabnejad, H., Pahl, C., Jamshidi, P., Estrada, G., 2017. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. CCGRID, IEEE, pp. 64–73. http://dx.doi.org/10.1109/CCGRID.2017.15.

Arcaini, P., Riccobene, E., Scandurra, P., 2015. 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, IEEE, pp. 13–23. http://dx.doi.org/10.1109/SEAMS.2015.10.

Aziz, Z.A., Abdulqader, D.N., Sallow, A.B., Omer, H.K., 2021. Python parallel processing and multiprocessing: A rivew. Acad. J. Nawroz Univ. 10 (3), 345–354. http://dx.doi.org/10.25007/ajnu.v10n3a1145.

Baarzi, A.F., Kesidis, G., 2021. Showar: Right-sizing and efficient scheduling of microservices. In: ACM Symposium on Cloud Computing (SoCC). pp. 427–441. http://dx.doi.org/10.1145/3472883.3486999.

Bakshi, K., 2017. Microservices-based software architecture and approaches. In: 2017 IEEE Aerospace Conference. IEEE, pp. 1–8. http://dx.doi.org/10.1109/AERO.2017.7943959.

Balla, D., Simon, C., Maliosz, M., 2020. Adaptive scaling of kubernetes pods. In: IEEE/IFIP Network Operations and Management Symposium. NOMS, IEEE, pp. 1–5. http://dx.doi.org/10.1109/NOMS47738.2020.9110428.

Baresi, L., Guinea, S., Leva, A., Quattrocchi, G., 2016. A discrete-time feedback controller for containerized cloud applications. In: 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 217–228. http://dx.doi.org/10.1145/2950290.295032.

Baresi, L., Quattrocchi, G., 2020. A simulation-based comparison between industrial autoscaling solutions and cocos for cloud applications. In: IEEE International Conference on Web Services. ICWS, IEEE, pp. 94–101. http://dx.doi.org/10.1109/ICWS49710.2020.00020.

Barna, C., Khazaei, H., Fokaefs, M., Litoiu, M., 2017. Delivering elastic containerized cloud applications to enable DevOps. In: IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, IEEE, pp. 65–75. http://dx.doi.org/10.1109/SEAMS.2017.12.

Choi, B., Park, J., Lee, C., Han, D., 2021. pHPA: A proactive autoscaling framework for microservice chain. In: 5th Asia-Pacific Workshop on Networking (APNet). pp. 65–71. http://dx.doi.org/10.1145/3469393.3469401.

Ding, Z., Huang, Q., 2021. COPA: A combined autoscaling method for kubernetes. In: IEEE International Conference on Web Services. ICWS, IEEE, pp. 416–425. http://dx.doi.org/10.1109/ICWS53863.2021.00061.

Dobies, J., Wood, J., 2020. Kubernetes Operators: Automating the Container Orchestration Platform. O'Reilly Media.

Dumpleton, G., 2018. Deploying to OpenShift: A Guide for Busy Developers. " O'Reilly Media, Inc.".

Fu, S., Mittal, R., Zhang, L., Ratnasamy, S., 2020. Fast and efficient container startup at the edge via dependency scheduling. In: 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). URL https://www.usenix.org/conference/hotedge20/presentation/fu.

Gheibi, O., Weyns, D., Quin, F., 2021. Applying machine learning in self-adaptive systems: A systematic literature review. ACM Trans. Auton. Adapt. Syst. 15 (3), 1–37. http://dx.doi.org/10.1145/3469440.

Gias, A.U., Casale, G., Woodside, M., 2019. ATOM: Model-driven autoscaling for microservices. In: IEEE 39th International Conference on Distributed Computing Systems. ICDCS, IEEE, pp. 1994–2004. http://dx.doi.org/10.1109/ICDCS.2019.00197.

Google, 2024. Microservice benchmark application: Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo. (Accessed 10 March 2024).

Horovitz, S., Arian, Y., 2018. Efficient cloud auto-scaling with SLA objective using Q-learning. In: IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, pp. 85–92. http://dx.doi.org/10.1109/FiCloud.2018.00020.

Hossen, M.R., Islam, M.A., 2022. A lightweight workload-aware microservices autoscaling with qos assurance. http://dx.doi.org/10.48550/arXiv.2202.00057, ArXiv E-Prints, Pp. ArXiv–2202.

Imdoukh, M., Ahmad, I., Alfailakawi, M.G., 2020. Machine learning-based auto-scaling for containerized applications. Neural Comput. Appl. 32, 9745–9760. http://dx.doi.org/10.1007/s00521-019-04507-z.

Islam, S., Keung, J., Lee, K., Liu, A., 2012. Empirical prediction models for adaptive resource provisioning in the cloud. Future Gener. Comput. Syst. 28 (1), 155–162. http://dx.doi.org/10.1016/j.future.2011.05.027.

Jayalath, R.K., Ahmad, H., Goel, D., Syed, M.S., Ullah, F., 2024. Microservice vulnerability analysis: A literature review with empirical insights. IEEE Access http://dx.doi.org/10.1109/ACCESS.2024.3481374.

Joshi, N.S., Raghuwanshi, R., Agarwal, Y.M., Annappa, B., Sachin, D., 2023. ARIMA-PID: container auto scaling based on predictive analysis and control theory. Multimedia Tools Appl. 1–18. http://dx.doi.org/10.1007/s11042-023-16587-0.

Ju, L., Singh, P., Toor, S., 2021. Proactive autoscaling for edge computing systems with Kubernetes. In: 14th IEEE/ACM Int. Conf. on Utility and Cloud Computing Companion. UCC, pp. 1–8. http://dx.doi.org/10.1145/3492323.349558.

Kang, P., Lama, P., 2020. Robust resource scaling of containerized microservices with probabilistic machine learning. In: IEEE/ACM 13th International Conference on Utility and Cloud Computing. UCC, IEEE, pp. 122–131. http://dx.doi.org/10.1109/UCC48980.2020.00031.

Karn, R.R., Das, R., Pant, D.R., Heikkonen, J., Kanth, R., 2022. Automated testing and resilience of microservice's network-link using istio service mesh. In: 31st Conference of Open Innovations Association. FRUCT, IEEE, pp. 79–88. http://dx.doi.org/10.23919/FRUCT54823.2022.9770890.

Khazaei, H., Ravichandiran, R., Park, B., Bannazadeh, H., Tizghadam, A., Leon-Garcia, A., 2017. Elascale: autoscaling and monitoring as a service. http://dx.doi.org/10.48550/arXiv.1711.03204, arXiv preprint arXiv:1711.03204.

Liu, B., Buyya, R., Nadjaran Toosi, A., 2018. A fuzzy-based auto-scaler for web applications in cloud computing environments. In: 16th International Conference on Service-Oriented Computing. ICSOC, Springer, pp. 797–811. http://dx.doi.org/10.1007/978-3-030-03596-9_57.

Locust, 2024. An open-source load testing tool. https://locust.io. (Accessed 28 March 2024).

Luo, S., Lin, C., Ye, K., Xu, G., Zhang, L., Yang, G., Xu, H., Xu, C., 2024. Optimizing resource management for shared microservices: A scalable system design. ACM Trans. Comput. Syst. 42 (1–2), 1–28. http://dx.doi.org/10.1145/3631607.

Marie-Magdelaine, N., Ahmed, T., 2020. Proactive autoscaling for cloud-native applications using machine learning. In: IEEE Global Communications Conference. GLOBECOM, pp. 1–7. http://dx.doi.org/10.1109/GLOBECOM42002.2020.9322147.

Marques, G., Senna, C., Sargento, S., Carvalho, L., Pereira, L., Matos, R., 2024. Proactive resource management for cloud of services environments. Future Gener. Comput. Syst. 150, 90–102. http://dx.doi.org/10.1016/j.future.2023.08.005.

Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., Kim, S., 2020. Horizontal pod autoscaling in kubernetes for elastic container orchestration. Sensors 20 (16), 4621. http://dx.doi.org/10.3390/s20164621.

Nguyen, H.X., Zhu, S., Liu, M., 2022. Graph-PHPA: graph-based proactive horizontal pod autoscaling for microservices using LSTM-GNN. In: IEEE 11th Int. Conf. on Cloud Networking. pp. 237–241. http://dx.doi.org/10.1109/CloudNet55617.2022.9978781.

Nitto, E.D., Florio, L., Tamburri, D.A., 2020. Autonomic decentralized microservices: The Gru approach and its evaluation. Microserv.: Sci. Eng. 209–248. http://dx.doi.org/10.1007/978-3-030-31646-4_9.

Nouri, S.M.R., Li, H., Venugopal, S., Guo, W., He, M., Tian, W., 2019. Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. Future Gener. Comput. Syst. 94, 765–780. http://dx.doi.org/10.1016/j.future.2018.11.049.

Persico, V., Grimaldi, D., Pescape, A., Salvi, A., Santini, S., 2017. A fuzzy approach based on heterogeneous metrics for scaling out public clouds. IEEE Trans. Parallel Distrib. Syst. 28 (8), 2117–2130. http://dx.doi.org/10.1109/TPDS.2017.2651810.

Prophet, 2024. ML-based time series forecasting model. Available at: https://facebook.github.io/prophet. (Accessed 16 March 2024).

Qiu, H., Banerjee, S.S., Jha, S., Kalbarczyk, Z.T., Iyer, R.K., 2020. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In: 14th USENIX Symposium on Operating Systems Design and Implementation. pp. 805–825, URL https://dl.acm.org/doi/10.5555/3488766.3488812.

Qu, C., Calheiros, R.N., Buyya, R., 2018. Auto-scaling web applications in clouds: A taxonomy and survey. ACM Comput. Surv. 51 (4), 1–33. http://dx.doi.org/10.1145/3148149.

Rossi, F., Cardellini, V., Presti, F.L., 2020a. Hierarchical scaling of microservices in kubernetes. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems. ACSOS, IEEE, pp. 28–37. http://dx.doi.org/10.1109/ACSOS49614.2020.00023.

Rossi, F., Cardellini, V., Presti, F.L., 2020b. Self-adaptive threshold-based policy for microservices elasticity. In: 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. MASCOTS, IEEE, pp. 1–8. http://dx.doi.org/10.1109/MASCOTS50786.2020.9285951.

Rossi, F., Cardellini, V., Presti, F.L., Nardelli, M., 2020c. Geo-distributed efficient deployment of containers with Kubernetes. Comput. Commun. 159, 161–174. http://dx.doi.org/10.1016/j.comcom.2020.04.061.

Rossi, F., Cardellini, V., Presti, F.L., Nardelli, M., 2022. Dynamic multi-metric thresholds for scaling applications using reinforcement learning. IEEE Trans. Cloud Comput. http://dx.doi.org/10.1109/TCC.2022.3163357.

Rossi, F., Nardelli, M., Cardellini, V., 2019. Horizontal and vertical scaling of container-based applications using reinforcement learning. In: IEEE 12th International Conference on Cloud Computing. CLOUD, IEEE, pp. 329–338. http://dx.doi.org/10.1109/CLOUD.2019.00061.

Roy, N., Dubey, A., Gokhale, A., 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In: IEEE 4th International Conference on Cloud Computing. IEEE, pp. 500–507. http://dx.doi.org/10.1109/CLOUD.2011.42.

Santos, J., Wauters, T., Volckaert, B., De Turck, F., 2023. Gym-hpa: Efficient auto-scaling via reinforcement learning for complex microservice-based applications in Kubernetes. In: NOMS IEEE/IFIP Network Operations and Management Symposium. IEEE, pp. 1–9. http://dx.doi.org/10.1109/NOMS56928.2023.10154298.

da Silva, T.P., Neto, A.F.R., Batista, T.V., Lopes, F.A., Delicato, F.C., Pires, P.F., 2021. Horizontal auto-scaling in edge computing environment using online machine learning. In: 2021 IEEE Int. Conf. on Dependable, Autonomic and Secure Computing, Int. Conf. on Pervasive Intelligence and Computing, Int. Conf. on Cloud and Big Data Computing, Int. Conf. on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). pp. 161–168. http://dx.doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00038.

Soppelsa, F., Kaewkasi, C., 2016. Native docker clustering with swarm. Packt Publishing Ltd.

Subramaniam, A., Subramaniam, A., 2023. Automated resource scaling in Kubeflow through time series forecasting. In: 5th Int. Conf. on Cybernetics, Cognition and Machine Learning Applications. ICCCMLA, pp. 173–179. http://dx.doi.org/10.1109/ICCCMLA58983.2023.10346870.

Toka, L., Dobreff, G., Fodor, B., Sonkoly, B., 2021. Machine learning-based scaling management for kubernetes edge clusters. IEEE Trans. Netw. Serv. Manag. 18 (1), 958–972. http://dx.doi.org/10.1109/TNSM.2021.3052837.

Tong, J., Wei, M., Pan, M., Yu, Y., 2021. A holistic auto-scaling algorithm for multi-service applications based on balanced queuing network. In: IEEE International Conference on Web Services. ICWS, IEEE, pp. 531–540. http://dx.doi.org/10.1109/ICWS53863.2021.00074.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J., 2015. Large-scale cluster management at Google with Borg. In: Tenth European Conf. on Computer Systems (EuroSys). pp. 1–17. http://dx.doi.org/10.1145/2741948.2741964.

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. 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. Springer, pp. 76–107. http://dx.doi.org/10.1007/978-3-642-35813-5_4.

Yang, J., Liu, C., Shang, Y., Cheng, B., Mao, Z., Liu, C., Niu, L., Chen, J., 2014. A cost-aware auto-scaling approach using the workload prediction in service clouds. Inf. Syst. Front. 16, 7–18. http://dx.doi.org/10.1007/s10796-013-9459-0.

Yang, Z., Nguyen, P., Jin, H., Nahrstedt, K., 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In: IEEE 39th International Conference on Distributed Computing Systems. ICDCS, IEEE, pp. 122–132. http://dx.doi.org/10.1109/ICDCS.2019.00021.

Yu, G., Chen, P., Zheng, Z., 2020. Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. IEEE Trans. Cloud Comput. 10 (2), 1100–1116. http://dx.doi.org/10.1109/TCC.2020.2985352.

ZargarAzad, M., Ashtiani, M., 2023. An auto-scaling approach for microservices in cloud computing environments. J. Grid Comput. 21 (4), 73. http://dx.doi.org/10.1007/s10723-023-09713-7,

Zhang, S., Wu, T., Pan, M., Zhang, C., Yu, Y., 2020. A-SARSA: A predictive container auto-scaling algorithm based on reinforcement learning. In: IEEE International Conference on Web Services. ICWS, IEEE, pp. 489–497. http://dx.doi.org/10.1109/ICWS49710.2020.00072.

Zhou, D., Chen, H., Shang, K., Cheng, G., Zhang, J., Hu, H., 2022a. Cushion: A proactive resource provisioning method to mitigate SLO violations for containerized microservices. IET Commun. 16 (17), 2105–2122. http://dx.doi.org/10.1049/cmu2.12464.

Zhou, N., Zhou, H., Hoppe, D., 2022b. Containerization for high performance computing systems: Survey and prospects. IEEE Trans. Softw. Eng. 49 (4), 2722–2740. http://dx.doi.org/10.1109/TSE.2022.3229221.