Containerized Microservices: A Survey of Resource Management Frameworks

Lamees M. Al Qassem, Thanos Stouraitis, *Life Fellow, IEEE*, Ernesto Damiani, *Senior Member, IEEE*, and Ibrahim M. Elfadel, *Life Senior Member, IEEE*

***Abstract*—The growing adoption of microservice architectures (MSAs) has led to major research and development efforts to address their challenges and improve their performance, reliability, and robustness. Important aspects of MSA that are not sufficiently covered in the open literature include efficient cloud resource allocation and optimal power management. Other aspects of MSA remain widely scattered in the literature, including cost analysis, service level agreements (SLAs), and demand-driven scaling. In this article, we examine recent cloud frameworks for containerized microservices with a focus on efficient resource utilization using auto-scaling. We classify these frameworks on the basis of their resource allocation models and underlying hardware resources. We highlight current MSA trends and identify workload-driven resource sharing within microservice meshes and SLA streamlining as two key areas for future microservice research.**

***Index Terms*—Microservices, containers, resource management, container orchestration, machine learning, workload forecasting, reactive allocation, predictive allocation.**

I. Introduction

MICROSERVICES are a cloud native architectural style that consists of loosely coupled, independently deployable, smaller components and services. The microservice framework is a relatively new approach to cloud virtualization [1]. Microservices are the building blocks of an enterprise cloud application and are typically deployed in containers. In recent years, cloud service providers have been progressively adopting the containerized microservice approach as an alternative to the traditional monolithic virtualization solution based on virtual machines (VMs). Containerized MicroService Architectures (MSAs) provide a flexible and efficient way to allocate and distribute hardware resources throughout a cloud ecosystem. Unlike VMs, containers share the same operating system, eliminating the need for a hypervisor, thus reducing CPU and storage overhead. In this paper, we refer to containerized microservices as microservices for simplicity.

Manuscript received 18 September 2023; revised 9 February 2024; accepted 10 April 2024. Date of publication 15 April 2024; date of current version 21 August 2024. Funding of this research was provided by the Khalifa University Center of Secure Cyberphysical System. The associate editor coordinating the review of this article and approving it for publication was S. Scott-Hayward. *(Corresponding author: Ibrahim M. Elfadel.)*

The authors are with the Center for Secure Cyber Physical Systems and the College of Computing and Mathematical Sciences, Khalifa University, Abu Dhabi, UAE (e-mail: lamees.alqassem@ku.ac.ae; thanos.stouraitis@ku.ac.ae; ernesto.damiani@ku.ac.ae; ibrahim.elfadel@ku.ac.ae).

Digital Object Identifier 10.1109/TNSM.2024.3388633

Despite its many advantages, the transition from monolithic to microservice architecture poses serious challenges to cloud service providers. For example, managing end-to-end tail latency (i.e., 95th percentile latency) of requests streaming through the microservice mesh is a recurrent problem of microservice platforms, which can result in a negative customer experience and loss of revenue for cloud providers [2]. Efficient resource management policies that meet system-wide Service Level Agreements (SLAs) are difficult to find and, when found, are computationally burdensome. Even leading cloud service providers such as Google, Amazon, and Alibaba have been suffering from inefficient resource utilization ($\leq$40%) as reported in recent studies [3], [4], [5]. Resource management techniques are far from optimal due to the distributed and dynamic nature of microservice systems [6]. Furthermore, complex interactions among microservices introduce new challenges in managing shared hardware resources and identifying critical resource bottlenecks [7]. In addition, the performance of containerized microservices can unexpectedly deteriorate when running on a cluster of virtual machines in the cloud [8], [9]. Due to these challenges, managing microservice-based applications and the underlying hardware resources is still more an art than a science.

Still, MSA shows great potential for optimization due to the availability of fine-grained performance data. When embodied in containers, MSA often exposes internals of the cloud service, such as resource utilization and performance metrics, which may enable fine-grained policies and, eventually, the use of Machine Learning models [10] to improve resource allocation, reduce power consumption, and achieve higher throughput while maintaining SLA compliance. This is particularly important for small to medium cloud service providers, for whom SLA compliance management is an overriding business requirement [11].

A large and growing number of studies are available to describe the challenges, concerns, and practices related to MSA [12], [13], [14], [15], [16], [17], [18]. These studies focus on the architectural aspects, design principles, and characteristics of microservices. For example, [15] is an analytical study of gray literature dedicated to the gains and pains of microservice-based solutions in the industry. Although linking containers to microservices seems obvious at the deployment stage of cloud-based solutions [19], existing literature rarely addresses the hardware management and resource scaling aspects of containerized microservices. An important underlying assumption of all these studies is that each microservice is deployed as a single container, which

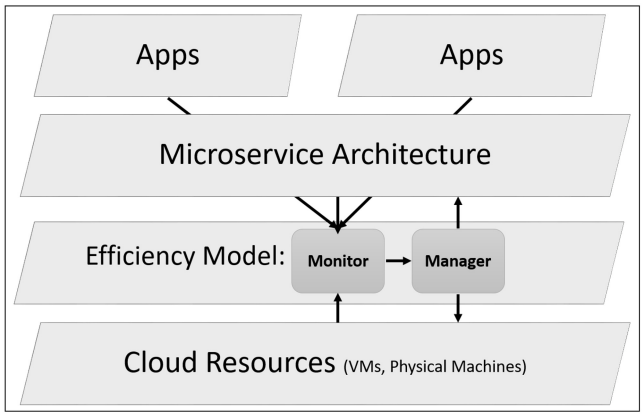

Fig. 1. The full stack of a microservice architecture with a resource manager.

facilitates the development of MSA scaling policies. As stated in [20], the main reason for adopting MSA is that it enables the scalability and maintainability of the cloud application code. Therefore, the present paper is focused on a survey of the various frameworks that are used to manage the scalability of containerized microservices.

Given the crucial role of containers in our microservice scalability survey, it is important to briefly review the recent literature specific to container technology. Representative publications of this literature include [21], [22], [23] which focus on container orchestrations as cloud virtualization techniques, regardless of virtualized software applications. In particular, [22] focuses on the container as the backbone of a lightweight cloud virtualization technology that can smoothly run services as varied as IoT, e-Commerce, and data warehousing. It further addresses container scheduling techniques, with an emphasis on the role that mathematical modeling plays. To the best of our knowledge, the important aspect of container resource utilization at microservice runtime has not been addressed in this body of literature. Nor has the equally important aspect of SLA-driven container management.

The main goal of our work is to provide a comprehensive synthesis of containerized microservice management frameworks, with a particular focus on efficient SLA-driven resource management. The present review article is also meant to bridge the gap between the container technology community and the cloud service community by focusing on the hardware management and scaling of containerized microservices with the objective of avoiding SLA violations. Specifically, our analysis of existing containerized microservice frameworks should prove valuable to both communities and beyond them, to academic researchers and cloud service providers.

Fig. 1 shows the complete stack of an MSA, comprising distinct layers crucial to the efficient functioning of the system. The top layer, "Apps", consists of different applications that comprise the microservice ecosystem. The apps are designed to be modular and independent. They interact seamlessly within the lower layer, Microservice Architecture, which has the core components that enable communication and orchestration. Each microservice runs independently while contributing to the system's comprehensive functionality. The interactions within the Microservice Architecture Layer involve service discovery, load balancing, and inter-service communication, providing an agile and responsive orchestration environment. The layer below is the resource management layer that supports the Microservice Architecture layer by providing it with the needed resources. The resource management layer comprises two main components: the resource monitor and the controller/manager. The monitor collects resource usage statistics from each running microservice. The controller manages the allocation and configuration of various replicas of microservice instances. This layer is meant to operate dynamically, with the monitor tracking the resource usage of workloads with unknown requirements so that the controller can allocate sufficient resources to run them. Throughout the paper, we will use the terms "controller" and "manager" interchangeably. Finally, the 'Cloud Resources' layer encompasses virtual machines (VMs) and physical machines, representing the underlying infrastructure supporting MSA. The dynamic allocation decisions made by the 'Manager' directly influence the utilization of these cloud resources.

Note on the Survey Methodology: In this article, we review resource management frameworks for containerized microservices. One key ingredient of such frameworks is the sharing of microservice platforms. The methodology we have adopted for our survey is as follows. We first conducted a bibliographic search based on the following keywords: *microservice framework, resource efficiency, resource management framework, microservice resource allocation, microservice energy efficiency, microservice power consumption, microservice cluster manager, microservice scaling, and microservice scheduling.* The search covered the period from the beginning of 2015 to the end of 2023. Based on this search, we have collected more than 100 articles, which we have pruned down to 65. The exclusion criteria for the collected articles are for platforms whose microservices are not containerized and for MSA solutions that do not address hardware resources or the vertical scaling level of the containerized microservices. On the other hand, we have limited the selected papers to those that explicitly contain the word "microservice" in the title and use containerization technologies such as Docker. We focus on platforms that use Docker containers as they are widely adopted in cloud computing. The selected articles were then classified into three categories based on the resource allocation strategy. These three categories are labeled "reactive," "predictive," and "hybrid." In each category, we have considered the following attributes:

1) The microservice framework and its management components.
2) The target hardware resources, along with their sharing and allocation policies among various microservices.
3) The test environment and the baseline design that are used to evaluate the models.
4) The main SLA requirements that the microservice is supposed to meet.
5) Target workloads and cloud applications.

Analyzing the above attributes helps compare the different MSA-based efficiency platforms, as will be illustrated in Tables I through V.

In summary, our contributions in this article are as follows:

1) We conduct an extensive survey of Microservices Architecture (MSA) resource management frameworks, offering a thorough examination of their diverse features, strengths, and limitations.
2) We survey MSA resource management frameworks from the viewpoint of their impact on improving resource utilization and reducing the cost of data center ownership.
3) We classify MSA resource management frameworks according to their decision-making paradigm. This approach provides a structured framework for understanding the underlying principles guiding the resource management strategies employed by these frameworks.
4) We provide a detailed analysis and comparison of the MSA resource management frameworks. The comparative analysis is instrumental in guiding practitioners and researchers in selecting the most suitable resource management framework based on their specific requirements and priorities.
5) We highlight future developments needed to improve MSA performance and efficiency. By pinpointing areas that require further research and development, we contribute insights that can drive innovation in the field, fostering improvements in MSA performance and resource efficiency.

The paper is organized as follows. A background on MSA and SLA is given in Section II. The classification of microservice resource management frameworks is explained in Section III. The various resource management frameworks are discussed in Sections V, IV, and VI. Detailed analysis and comparisons of the platforms surveyed are given in Section VII and a discussion of the MSA efficiency is given in Section VIII. The paper concludes with Section IX.

II. Background

A. Microservice Architecture (MSA)

A microservice architecture is typically composed of several loosely coupled services that work together in concert. Such an architecture allows for more agile development, shorter time to market, independent component updates, and easier service scale-up [24].

The MSA is a cloud-native architectural style in which an application is built as a set of loosely coupled services of fine granularity that communicate via thin Application Programming Interfaces (APIs). Given the fine granularity, each service is called a *microservice*. The microservice provides a specific business outcome and usually runs inside a container.

Containers are the software building blocks of containerized MSA. They bundle their own software, libraries, and configuration files but share the services of a single operating system kernel, requiring fewer resources than virtual machines [25]. Users can run various applications in distinct containers on the same host. Such containers can freely move between host machines, so developers can build, manage, and secure their applications without worrying about the host and network infrastructure. Using containers in the cloud has many well-known benefits [26]

1) It provides a uniform cloud platform for testing and deploying cloud workloads.
2) It enables fine-grained allocation of cloud resources.
3) It facilitates the continuous update of microservices.
4) It supports various languages and deployment platforms.

As a result of these advantages, cloud services are increasingly using containers as their main deployment units, while insuring that for a given workload, enough instances are always running and are being co-located according to their memory and CPU demands.

B. Service Level Agreement (SLA)

SLA, terms and conditions, privacy policy, and End User License Agreement (EULA) all refer to the same thing under different labels. An SLA is a legal contract between the service provider and the tenant that defines both the functional and contractual aspects of a service [27]. It includes, among several other clauses, the expected Quality of Service (QoS) metrics [28]. Performance, availability, cost, and latency are examples of QoS metrics. Each of these metrics has multiple QoS figures of merit (FoM). Performance, for example, is expressed by one or several FoMs, such as response time or throughput.

In general, SLAs consist of Service Level Objectives (SLOs) that must be met and are generally negotiated between tenants and the service provider [29]. These SLOs usually provide the target values of the QoS metrics and the thresholds used to quantify the status of the microservice. A complete survey on the formulation of SLA can be found in [11].

III. Classification of MSA Resource Management Frameworks

Despite their numerous advantages, containerized microservices pose significant challenges in cloud deployment and management. For one thing, the interdependency among microservices often results in cascading QoS violations that are not amenable to prompt detection and rapid recovery. For another, the sheer number of services used by microservice-based applications (up to thousands of services) under management in a given cloud cluster and the long-term distribution of this number across applications [30] are major obstacles to the execution of management policies designed for the monolithic cloud paradigm. Furthermore, the implementation, monitoring, and control infrastructures of these numerous microservices require a high level of automation [12]. To facilitate such automation, one possibility is to adopt a model-based approach in which a microservice resource management framework is designed and integrated as part of the container service. The model-based approach will benefit from the fact that most microservice frameworks usually have an efficient resource management model as part of their structural design and functional performance.

In general, efficient resource allocation is obtained when resource supplies are correctly matched to workload demands in a timely manner. The matching policy allows for a classification of existing resource management frameworks into three categories:

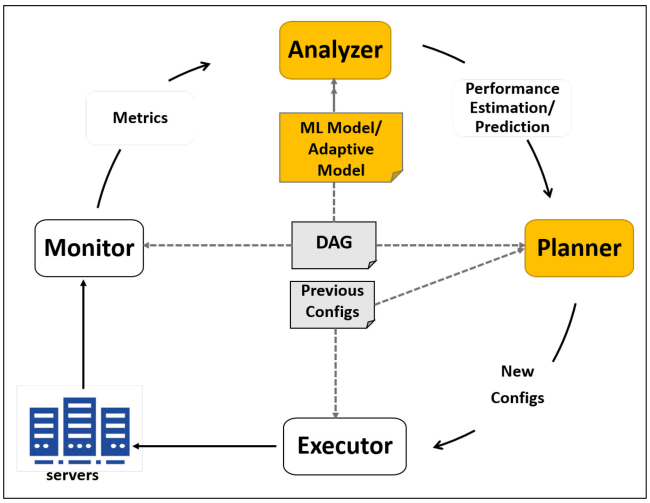

Fig. 2. The major components of the resource management framework along with their information flow.

1) **Reactive**: In this category, resources are dynamically allocated in response to system workload and network traffic conditions.
2) **Predictive**: In this category, future resource usage is predicted based on current usage and the type of workload.
3) **Hybrid**: In this category, reactive and predictive solutions are combined to reap the benefits of both approaches.

A cloud service management layer has a regular design pattern [7], [31], [32], [33] that generally consists of the following components: a monitor, an analyzer, a planner, and an executor. These components work together to allocate resources efficiently while maintaining SLA compliance. The components of resource management frameworks are illustrated in Fig. 2. The monitor collects metric data; the analyzer estimates or predicts performance based on the collected data; the planner sets new configurations; and finally, the executor updates the configurations of the hardware resources if and when needed. The main differentiators in the resource management framework are in the analyzer, the planner (both color-coded yellow in Fig. 2), and their implementations. For instance, in reactive systems, the analyzer is an adaptive model (e.g., the queueing model in [33]), while in predictive systems it is an inference model (e.g., the Gaussian regressor in [7]). The hybrid system contains both an adaptive model and an inference model. The planner has one or more of the following four parts: an autoscaler, a scheduler, a task co-locator, and a cluster/resource manager.

In the following three sections, we review the details of the three resource management frameworks (reactive, predictive, and hybrid) and discuss the advantages and limitations of each one.

IV. Reactive Frameworks

Reactive frameworks are event-driven models that provide the required computing resources based on observed resource utilization (e.g., CPU, memory, disk, and network I/O). These models assume that there is always a constant percentage of spare resources reserved to hedge against unexpected increases in resource demand. In other words, they dynamically allocate resources to microservices based on cloud workload and network traffic. The reactive frameworks will be discussed according to the four categories listed in Table I.

TABLE I
Comparison of Reactive Microservice Resource Management Frameworks

Category	Rule-based	[34], [35], [36], [38], [40]	
	Optimization-based	[41], [1], [42], [43], [44], [45], [46]	
	Task Co-location	[47], [48], [49]	
	Bidirectional AS	[50], [51], [33]	
Components	RM	[41], [1], [42], [33], [36]	
	Schd	[45]	
	AS	Vertical	[43]
		Horizontal	[44], [38], [40]
		Bidirectional	[50], [51], [33]
	CL	[47], [48], [49]	
Hardware	CPU	All reactive platforms	
	MEM	[36], [50], [43], [44], [45], [47], [49], [51], [52], [46]	
	Network	[36], [50], [47]	
	I/O	[36]	
SLAs	Latency	[41], [47], [48], [49], [38], [40], [48]	
	Cost	[38], [40], [44]	
	Throughput	[45], [49]	
	Reliability	[41]	
Workload	LC	[1], [33], [47], [48], [49],[46]	
	Batch	[35], [34], [49]	
	General	[41], [1], [38], [40], [36], [43], [45]	

A. Rule-Based Frameworks

The frameworks in this category are simple rule-based solutions that change hardware resources and scale containers when an SLA violation occurs or one of their predefined conditions is met. A dynamic CPU resource allocation framework for containerized microservices is presented in [34], [35]. Their aim is to reduce the over-utilization of CPU resources. This is accomplished by collecting runtime CPU utilization and making decisions only when over-utilization occurs. The model is a simple round-robin time-slicing policy that schedules one container at a time. Additionally, each container uses all hardware resources on the host machine for a given time interval. The size of the data collection window is an important implementation parameter, as it strongly affects the delay in response time. This one-container implementation is simple, but is unable to manage a complex application that has several containers running in parallel. Furthermore, it does not account for the microservice SLA requirements.

Mesos [36] is a widely used resource manager that can run on different distributed computing platforms such as Hadoop and Message Passing Interface (MPI). Its default allocation strategy is the Dominant Resource Fairness (DRF) allocation policy proposed in [37]. DRF is a fair allocation policy for multiple types of resource (e.g., CPU, memory, network I/O) to users with heterogeneous demands. DRF satisfies four main allocation properties that are considered important in a multiresource environment with heterogeneous requests:

1) Incentives for Sharing: DRF encourages users to share resources by ensuring that no user benefits if resources are distributed equally among them.
2) Immunity to deception: Users cannot improve their allocations by lying about their resource requirements.
3) Pareto optimality: Increasing the allocation for one user will result in decreasing the allocation for another user.

4) Freedom from envy: A user cannot prefer the allocation of another user.

Another well-known container-based reactive scaling framework is Kubernetes (K8s) [38]. It is a simple horizontal autoscaler that tracks the average CPU utilization and scales containers into or out of the cluster to reach the CPU/memory utilization threshold specified by the end user. This is in contrast to vertical scaling, where the attributes of the resource (e.g., CPUs, RAM) are modified across the cluster nodes. Kubernetes horizontal scaling has two management policies [39]:

1) *PodFitResources*: The total amount of requested resources should not exceed the capacity of the physical machine.
2) *LeastRequestPriority*: Allocate containers in machines with the least resources consumed.

Both policies aim to distribute containers across all machines in the cluster, but they cannot manage the scaling level of containers. K8s also offers a beta API for auto-scaling based on multiple metrics, with the actual scaling executed based on the metric with the largest scale. Therefore, using K8s to scale microservices that use a mix of resources could lead to a longer response time and more SLA violations. This is because the K8s autoscaler calculates the required number of replicas for each resource separately and performs scaling based on the largest value. Such calculations could lead to resource over-utilization. Furthermore, increasing the number of replicas does not necessarily imply the avoidance of SLA violations. This is, for instance, the case where all replicas suffer from out-of-memory (OOM) errors.

K8s uses the following formula to assign containers to a microservice:

$$desiredReplicas = ceil\left(currentReplicas * \left[\frac{currentMetricValue}{desiredMetricValue}\right]\right) \quad (1)$$

If the *desiredMetricValue* for CPU utilization is set to 40% and the *currentMetricValue* is 50%, the ratio is 1.2, which doubles the number of microservice replicas. In the case of multiple metrics, this calculation will be repeated for each metric and the largest value of the *desiredReplicas* will be selected. Scaling is performed only if the ratio in the formula is less than 0.9 or greater than 1.1. Defining resource utilization thresholds for different microservices in order to achieve the desired end-to-end tail latency at the application level is difficult, error-prone, and burdensome for application owners. In particular, setting unreasonable thresholds can result in resource over-provisioning or under-provisioning.

Docker Swarm [40], the first Docker container management system launched in 2014, is a native clustering system that manages a pool of Docker machines as if they were a single host. This enables users to horizontally scale containers in or out of a running cluster. Scaling commands, on the other hand, must be entered manually and are much too slow to respond to sudden load changes. Docker Swarm supports three types of node-ranking strategies: scatter, binpack, and random. However, it only supports a small number of provisional rules based on precise RAM and processor values from the available nodes. Furthermore, containers in Docker Swarm are viewed as independent, unrelated entities with no user preferences or workflow context information.

B. Optimization-Based Frameworks

Due to the large number of scales, the heterogeneity of nodes and their attributes, and the high variance of workloads, optimal resource allocation is a combinatorially complex problem in cloud management [53]. It belongs to the class of NP-complete problems and is best addressed using meta-heuristics [1]. According to [53], microservice resource management and optimization are topics whose exploration has just recently begun. For example, existing container management systems like Kubernetes and Docker Swarm use simple auto-scaling policies that are not optimal in any way. This leaves ample room for further development.

Now we review some recent publications that have proposed optimization approaches for efficient microservice resource allocation. Hoenisch et al. [43] propose optimization strategies for cloud deployment of traditional applications. Their proposed strategy is a multi-objective optimal virtualization model that scales VMs using a bidirectional policy (vertically and/or horizontally) to meet the computational requirements of their containers. The objectives of the model are cost, deployment time, and available resources. Among several usage modes, the model is used to assign a given container to the most appropriate VM that meets its computational requirements. The proposed model was compared with two well-known baseline VM autoscalers:

1) *One-for-All* where each VM is a quad-core CPU and hosts one container of each type per core.
2) *One-for-Each* where each VM is a single-core CPU that hosts only one container.

Both autoscalers implement threshold-based policies that lease or re-lease VMs when a certain threshold has been crossed. The cost of the proposed optimal solution is approximately 28% less than the two baselines. One possible research direction is to extend the approach of [43] to containers. This idea is developed in [44] where the focus is on elastic provisioning of VMs for containers. In this context, scaling is done at the VM-level and does not consider the scalability of containers. The scaling problem in [44] is formulated as an Integer Linear Programming problem that addresses only the heterogeneity of container requirements and virtual machine resources. The optimization problem considers two main objectives: (1) container deployment time and (2) cost of virtual machine leasing for container execution. Like [43], [44] has the potential to be extended to MSAs. Both [43] and [44] can be augmented to account for communications among containers to achieve better control over network latency.

On the other hand, Guerrero et al. [1], propose an optimal allocation policy that is geared towards MSAs. Their algorithm efficiently allocates containers and automatically scales them across different machines in a cluster. Given the complexity of the optimization problem, they use the heuristics of a genetic algorithm based on the non-dominated sorting genetic algorithm-II (NSGA-II) paradigm. The optimization

formulation considers four objectives: (1) uniform distribution of microservice workloads across their container replicas (threshold distance), (2) balancing the use of cluster resources, (3) reduction of network overheads (network distance), and (4) improvement of system reliability by avoiding single point of failure (system failure). The proposed solution was compared against K8s' allocation policies, with the proposed algorithm outperforming K8s while requiring fewer physical machines and, therefore, consuming less energy. The optimization algorithm of [1] seems to be the first to use a genetic algorithm to optimize the allocation of cluster containers. It can be further developed [1] by accounting for communication between microservices and including the dependence of cluster reliability on the number of microservice requests.

Vhatkar and Bhole [42] use the same four objectives as [1] but apply a different optimization technique. Their proposed algorithm is called velocity-updated gray wolf optimization and consists of a hybrid combination of particle swarm optimization and gray wolf optimization. Their system has been evaluated against other optimization techniques in terms of cost and statistical behavior. The comparison results show that the proposed hybrid algorithm can reduce the cost by approximately 27% compared to [1]. The main drawback of the model is that it does not account for the SLA of microservice applications. Moreover, a detailed comparison between the optimization methods of [42] and [1] would be valuable for both, especially since their objectives are the same.

Another optimization approach is the one proposed in [54], which is based on formalizing resource allocation as a bin-packing problem (BPP), an NP-hard optimization problem. A recent SLA-based resource allocation framework that uses a BPP optimization strategy is presented in [41]. The proposed solution aims to reduce operating costs for cloud brokers, who are responsible for containerizing the user's microservices and deploying them on a selected cloud infrastructure. The solution is based on the assumption that the microservices of any given application are independent but communicate with each other through a messaging interface deployed at the premises of the cloud broker. The solution of [41] has been compared with Docker's state-of-the-art deployment strategies (Spread, Random, and BinPack) and evaluated using three performance metrics: number of active physical machines, total cost outlay, and system reliability. The proposed solution has similar performance to Docker's BinPack but outperformed Spread and Random in two metrics: the number of active machines and cost.

In [45], Ant Colony Optimization (ACO) has been used to schedule containers across different hosts in Docker Swarm. The proposed algorithm is a modified version of SwarmKit [55], a round-robin scheduler developed by Docker for task scheduling and used by the Docker Swarm mode. According to the experimental results, the ACO-based scheduler has outperformed SwarmKit and made NGINX servers 15% faster.

ERMS [46] is a recent resource management system that certifies SLAs for shared microservice environments. It accurately models microservice latency as a piecewise linear function based on workload, microservice replicas, and CPU and memory resource usage. Based on this model, ERMS builds intelligent scaling strategies that optimize latency targets for complex microservice dependencies. ERMS further enhances efficiency through a novel scheduling policy that yields significant improvements. This policy prioritizes requests from various services and globally orchestrates resource scaling for all microservices. This policy reduces the number of container replicas by up to 50%. For evaluation, the ERMS prototype was implemented on top of K8s and evaluated using the DeathStarBench [56] microservice benchmark. The evaluation results demonstrate that ERMS significantly reduces the likelihood of SLA violations by 5X and achieves a reduction in resource usage by 1.6X compared with three previous policies: GrandSLAm [57], Rhythm [58], and Firm [59].

Given the complexity of the MSA resource optimization problem, heuristic methods have been proposed to find approximate solutions. These heuristics aim at achieving a wide variety of non-functional properties, such as fairness in resource allocation, load balancing, network traffic, and energy consumption. Most of them focus on system-oriented metrics such as hardware resource requirements, energy consumption, and the number of physical machines. Only a few consider user-oriented metrics such as deployment time, operational cost, and response time [1], [43], [44]. In terms of optimization techniques, these heuristic methods rely on population approaches such as genetic algorithms [1] and ant-colony optimization [45].

C. Task Co-Location

Many reactive systems introduce QoS-aware resource allocation models for latency-critical (LC) jobs and microservices. Their motivation is the increasing percentage of LC jobs (e.g., social media, Web search, webmail, machine translation, and online shopping) in the cloud workload. The resource usage of LC jobs is uneven due to diurnal cycles and unexpected fluctuations in user access. This leads to the under-utilization of allocated resources, especially during off-peak hours. Arbitrarily allocation of unused resources to other tasks, such as batch jobs, may not be optimal, as it may cause delays, thus violating the SLA of the LC job. Therefore, there is a need for SLA-aware co-location schedulers that are able to launch batch jobs on the same machine as the LC job to exploit unused resources without violating the SLA of the LC.

Heracles [47] is one of the first systems that addresses the aforementioned challenge of co-location scheduling. Heracles is a feedback-based controller that employs a combination of hardware and software isolation mechanisms (core isolation, last-level cache (LLC) isolation, power isolation, and network traffic isolation) to maximize server efficiency by co-locating one LC job with multiple batch jobs. The controller is designed to find the appropriate settings for all of these mechanisms for any LC workload and any collection of batch jobs. Heracles solves the controller design problem as an optimization problem, where the aim is to maximize resource utilization without SLA violations for the LC workload.

Heracles has been evaluated with various production-level LC and batch workloads in Google clusters, and 90% utilization has been achieved without SLA violations. However, the Heracles controller focuses on meeting the QoS of a single LC job while leaving best-effort batch workloads unmanaged.

PARTIES [48], has been designed to overcome the single-LC limitation of Heracles by allowing multiple LC jobs to share the same physical node with batch jobs while still meeting the SLA of each LC. To meet the latency goals of multiple co-located LC jobs, PARTIES incrementally changes the resource attributes (e.g., number of cores, memory space, etc.) of one LC job at a time while evaluating its observed output. This incremental process continues until the QoS of all LC jobs is achieved. The corresponding balance of resources is then allocated to the batch jobs. PARTIES is a significant development in terms of job co-location and resource allocation within a single physical machine, but it leaves open the problem of co-location and resource allocation across multiple physical nodes. Furthermore, PARTIES processes the batch jobs under the "best-effort" paradigm without maximizing their throughput.

The latter problem is precisely the one addressed in CLITE [49], which provides a comprehensive framework for resource partitioning to simultaneously meet the QoS of multiple co-located LC jobs while maximizing the throughput of the background batch jobs. To achieve its goals, CLITE uses Bayesian optimization to find near-optimal resource configurations for complex objective functions using a limited number of sample points in its search space. CLITE has been compared with PARTIES [48], Heracles [47] and a brute-force off-line search (ORACLE) that provides the optimal solution. The evaluation results demonstrated that CLITE has been able to co-locate several LC jobs close to the ORACLE golden solution at loads much higher than those of Heracles and PARTIES. Quantitatively, the CLITE solution is within 5% of ORACLE and outperforms PARTIES and Heracles by more than 15% in most scenarios.

All the above models are actually task co-locators that co-locate latency-critical tasks with batch jobs in a way that maximizes resource utilization without violating QoS. Two of the proposed systems (PARTIES and CLITE) use multiple resource partitioning, while others (e.g., Heracles) focus on single resource partitioning.

D. Bidirectional Autoscalers

All these studies follow classic scalability patterns originally designed for VMs, scaling containers either horizontally or vertically. More recent container management systems have emerged to scale containers both horizontally and vertically.

We call such systems bidirectional. Kwan et al. [51] propose HyScale, a bidirectional solution built on top of the Docker Engine without a container orchestration system. The proposed model consists of two rule-based autoscalers: HyScale_CPU and HyScale_CPU+MEM. HyScale_CPU scales the microservices along the horizontal and vertical directions separately, depending on the CPU utilization. It prioritizes vertical scaling and uses horizontal scaling only when there are insufficient resources on the host machine to meet workload demands. HyScale_CPU+MEM is a multi-metric version of HyScale_CPU that scales containers based on both CPU and memory utilization. The proposed autoscalers have been compared with the classic K8s horizontal autoscaler in terms of SLA violations, resource utilization, and user-perceived performance. The latter is expressed in terms of the average response time to the user and the number of failed requests. In the CPU-bound experiments, the HyScale_CPU+MEM has the lowest response time, while HyScale_CPU has the lowest number of failed requests. Both algorithms have outperformed K8s under steady-state and transient CPU loads. In the mixed CPU- and memory-bound experiments, HyScale_CPU+MEM has the fastest response time with almost zero failed requests, while HyScale_CPU has more limited performance due to a preference for vertical over horizontal scaling in the absence of memory requests. Regarding SLA violations, both HyScale algorithms have outperformed K8s in CPU-bound experiments. Finally, the efficiency of the algorithm is calculated using the response-time-resource efficiency (RTE) metric, which is defined as the ratio of the response time to the average resource usage. A lower RTE is preferable because it means that with each unit of resources used, less time is wasted waiting for a response. In the experiments, K8s has the highest RTE, while HyScale_CPU+MEM has the lowest. In summary, HyScale_CPU+MEM performs the best because it considers both CPU and memory resources when scaling the microservice containers. On the other hand, HyScale_CPU is outperformed by K8s due to the former's preference for vertical scaling.

Atom [33] is another bidirectional scaling system that combines the layered queuing network model with heuristic-based approximations. Atom uses a genetic algorithm in a time-bounded search to dynamically scale the number of microservice replicas and CPU sharing among containers. A quantitative evaluation has been applied to study how Atom performs during an increase in workload. The results indicate that the Atom bidirectional autoscaler outperforms the one-dimensional ones. In addition, the results show that if a service is amenable to both vertical and horizontal scaling, workload characteristics must be considered to determine which of the two scalings is preferable.

Finally, SHOWAR [50] is a recent bidirectional autoscaler that uses the empirical variance and the mean of historical usage to estimate the optimal resource for running microservices. It scales the microservices horizontally based on a given target latency to maintain a steady response time. The target latency is the 95^{th} percentile of the measured latency, averaged over one minute of observation. Then, the difference between the measured and target latency is used in a proportional-integral-derivative (PID) controller to compute the number of microservice replicas.

E. Discussion

The main goal of resource management is to meet the needs of end users by efficiently mapping workloads to resources. In general, this can be achieved dynamically by

either vertical scaling, where the amount of resources allocated for each container is changed, as in [34], [43], or horizontal scaling, where the number of running containers is changed, as in Kubernetes [38] and Docker Swarm [40]. Although these early systems adopted the same unidirectional (scale-up or scale-out) strategies as for VMs [60], recent approaches take full advantage of container manageability. Two systems [33], [51] have adopted a bidirectional approach to benefit from both the fine-grained resource management of vertical scaling and the high resource availability of horizontal scaling.[1] Most of the reviewed work in this category formulates the resource management problem as a multi-objective optimization problem and uses metaheuristic techniques to solve it. Such solutions tend to be computationally expensive, especially for large-scale problems, and, therefore, parallel and distributed implementations on HPC nodes are often needed to reduce execution time and avoid an increase in tail latency.

It is important to realize that containerization and orchestration industry solutions such as Kubernetes and Docker Swarm are reactive, and therefore allocate the required computational resources based on actual consumption (e.g., CPU, memory, disk, and network I/O utilization). They automatically scale containers based on user-defined, microservice-driven, static resource utilization thresholds. This puts a considerable burden on application owners who are concerned about tail latency. To determine a good scaling threshold, users should not only identify the relationship between the system metric (i.e., utilization) and the application metric (i.e., response time), but also characterize the CPU and/or memory bottlenecks of the application. Without predictive performance models, the process of setting appropriate thresholds for resource allocation to meet end-to-end tail latency requirements would be time consuming and prone to errors.

V. Predictive Frameworks

In the case of frequent, large-scale workload variations, the reactive resource allocation strategies will incur large delays between observations and responses and will result in non-optimal resource configurations. Predictive approaches can compensate for delays while facilitating convergence to near-optimal configurations. Both cloud service providers and tenants could reduce operational costs if resource usage is forecast accurately. Recent work on predictive resource allocation systems will be reviewed in this section. The discussion will be based on the four components listed in Table II.

A. Resource Management Frameworks

The first such system is EMARS [52], a resource management system for serverless platforms focused on efficient memory allocation for containers. It is based on a predictive model that allocates and limits memory as a function of application workload and resource usage. This policy is motivated by experimental evidence that has shown how latency increases as the allocated memory limit decreases. However, latency is not affected when the memory limit is extended beyond a "sweet spot" that can guarantee near-minimal latency. This shows that the memory allocation decision should be made based on application requirements rather than on default settings.

[1]Even though K8s offers a Vertical Pod Autoscaler service, it is very limited compared to its Horizontal Pod Autoscaler.

TABLE II
Comparison of Predictive Microservice Resource Management Frameworks

Components	RM	[53], [61], [62], [63], [64]	
	Schd	[65] [57]	
	AS	Vertical	-
		Horizontal	[7], [32], [31], [66], [67]
		Bidirectional	[68], [69], [70]
	CL	[67]	
Hardware	CPU	All predictive platforms except EMARS	
	MEM	[62], [65], [63], [67], [71], [68], [70]	
	Network	[61], [64], [65], [70]	
	I/O	[61], [62], [68]	
SLAs	Latency	[65], [52], [7], [57], [32], [66], [64], [68], [69], [70]	
	Cost	[69], [70]	
	Throughput	[57], [32], [68], [69]	
	Reliability	-	
Workload	LC	[61], [65], [7], [32], [69]	
	Batch	[57], [66]	
	General	[62], [31], [68]	

Sinan [61] is a data-driven resource manager for interactive ML-based microservices. The system consists of three main components: (1) an on-line centralized scheduler that periodically queries the distributed operators to obtain information about the CPU, memory, network, and I/O usage; the collected resource usage history is then passed to the ML models to predict the end-to-end latency; based on the model's output, the scheduler chooses the best allocation with the least resources while meeting the QoS; (2) distributed operators located on each server and acting as performance monitors; and (3) a performance forecaster that hosts the ML models. The ML model considers the current state of the system and the application workload to predict the end-to-end latency and the probability of QoS violations for a given resource configuration. These predictions help maximize resource efficiency, meet SLAs, and reduce server power consumption. Sinan has two ML models: (1) a CNN that predicts the end-to-end latency of the next decision interval and (2) a boosted tree that estimates the QoS violation probability in the long term. The system was evaluated using a microservice-based social network application [56] and compared with two traditional autoscaling policies [72]: AS_Opt that reduces frequency and cores when CPU utilization is less than 40% and 30%, respectively, and AS_Cons, which is more conservative, as it reduces frequency and cores when CPU utilization is less than 30% and 20%, respectively. The evaluation results show that [61] is capable of meeting the SLA of the system without sacrificing the efficiency of resource allocation.

B. Microservice Schedulers

GrandSLAm [57] is a microservice execution framework that aims to improve both the utilization of resources in the data center and the performance of the workload without violating the SLA of LC jobs. It increases resource utilization by sharing microservices among multiple applications in the most efficient way. The proposed framework estimates the completion time as requests propagate through multiple microservices using a linear regression model. It then uses the estimation model to reorder the requests based on

their computational slack and executes them in an Earliest Deadline First (EDF) policy, while batching multiple, less critical requests to the maximum extent possible to increase throughput without violating the critical latency constraints. The system has been evaluated against a baseline with FIFO request execution and no microservice sharing between multiple applications. Evaluation results have shown that the GrandSLAm framework achieves a 3X throughput improvement over the benchmark for a wide range of real-world AI and ML applications without SLA violations. The main drawback of the platform is the need to collect enough workload data to be able to estimate the completion time of the running application. Furthermore, the DAG diagram of the microservice mesh should be known for a better latency estimation.

C. Task Co-Location

Another approach to the efficient utilization of cluster resources without SLA violations is to deploy containers on the most appropriate node in a way similar to the reactive co-locators discussed in the previous section.

Tao et al. [62] proposes a container placement algorithm based on fuzzy inference (FI) to dynamically predict the node most suitable to host an application container. The entire system consists of two models: (1) a load balancer (LB) that consists of the FI model that predicts the utilization level of a node and selects the least utilized one that meets the minimum container requirements; and (2) a container manager (CM) that deploys containers based on the configuration file generated by LB. The system has been compared with the round-robin policy, and the results have shown that both approaches have similar performance when the number of parallel workflows is less than 4. As the number of workflows has increased, the FI model has performed better. When all resources are exhausted, RR or FI does not select a single node to host containers. One main limitation of the proposed FI algorithms is the lack of container sharing among various workflows. Consequently, the proposed FI placement algorithm is only effective with modest workloads.

Other predictive systems are based on task assignments to containers that are shared among various workloads. One such system is a Containerized Task Co-Location (CTCL) scheduler [65] for heterogeneous workloads, which does not require prior knowledge of the submitted tasks. The CTCL algorithm categorizes the tasks submitted based on their CPU and memory resource consumption using behavioral identification and an off-the-shelf K-mean++ clustering code. The system has been evaluated with the Container-CloudSim simulator [73] using Alibaba 2018 cluster traces [74] in terms of resource utilization and rescheduling costs. It has been compared with the original traces and Kubernetes [38] scheduling policies. Experiments have shown that CTCL can increase overall resource efficiency by 38% and reduce the rescheduling rate by 99%. Furthermore, CTCL outperforms Alibaba traces and Kubernetes policies in terms of resource efficiency. However, the proposed CTCL solution has two main limitations:

1) Memory utilization during the initial Empirical Observation Time (EOT) is lower than the original traces and K8s. As a result, task allocation is allowed by the CTCL scheduler only after the first round of behavioral recognition.
2) There is a trade-off between resource efficiency and classification accuracy, defined by EOT length. Therefore, to maximize system performance, a proper EOT setup should find a balance between behavior and workload prediction metrics.

D. Autoscalers

The authors of HyScale [51], discussed in the previous section, have also tried to predict future CPU utilization using neural networks (NNs) [31]. They have trained 4 NNs offline: a standard NN, Layer Normalized Long Short Term Memory (LN-LSTM), Multi-layered LN-LSTM, and Convolutional Multi-layered LN-LSTM. These pre-trained models serve as baselines and are reactively updated to reflect real-time data using an online technique proposed in [51]. As it turns out, the Convolutional Multi-layered LN-LSTM has the lowest error in the training phase and has been used in the online training as the reference. The updated online model outperformed its reference with a prediction error of about 3.77%. Online learning has proven to be a reliable method for identifying anomalies in time series. This is due to the ability of online learning to adapt to the most recent incoming data patterns.

RScale [7] is an autoscaler that provides an end-to-end performance guarantee. RScale uses a predictive model based on Gaussian Process (GP) regression to predict the end-to-end tail latency of microservice workflows and provides confidence bounds on each prediction with minimal overhead. RScale periodically collects resource usage metrics at two levels: container and VM levels. The periodically calculated utilization value is used as the threshold value in Eq. (1) to find the desired number of containers. The microservices are either scaled up by adding more containers or down by removing containers. The RScale evaluation results show that in a parametric scenario where the service delays are either normally distributed or independent from each other, the proposed system can meet the system tail latency requirement even in the presence of performance interference and evolving system dynamics. The recent emergence of non-stationary and even non-parametric scenarios has triggered an interest in using ML-based prediction techniques.

The latency of microservice applications is predicted in [7] and [61] based on the history of resource usage and latency. Unfortunately, predictive ML models are application-dependent and not readily applicable to arbitrary cloud workloads. Consequently, cloud service providers must adapt the ML model to each microservice application and run them first in a tuning mode based on the collected runtime data. Such application dependence and the need for tuning are the main limitations of latency predictive ML models.

A very recent predictive autoscaler has been proposed that employs two ML models, one for CPU usage and one for submitted requests, to predict the number of replicas needed

for each microservice [32]. The model also considers the impact of microservice scaling on other microservices under a given workload. The experimental results indicate that the proposed solution outperforms the K8s horizontal autoscaler in terms of response time and throughput. Moreover, it takes fewer actions to achieve a desirable efficiency and a QoS target for a given workload. Even though this autoscaler outperforms the horizontal autoscaler of K8s, it is still a threshold-based model.

Abdullah et al. [66] propose a predictive ML-based horizontal autoscaling model that uses collected workload observations to predict future workload bursts. This predicted value is used to find the number of containers needed to meet the latency SLA requirement. Different state-of-the-art ML models were evaluated, including Linear Regression, Polynomial Regression, Elastic Net Regression, XGBoost Regression, Random Forests Regression, Decision Tree Regression (DTR), Support Vector Regression, and Multi-layer Perceptron Regression. The model with the least MSE, namely, DTR was selected. The predictive model is designed for CPU-bound microservices. The Fast Fourier Transformation (FFT) algorithm [75] was used to evaluate system performance and compare it to four state-of-the-art autoscaling methods used for virtual machines. The proposed model outperformed the state-of-the-art VM autoscalers under various workloads.

As an alternative to workload prediction, threshold learning has been proposed in [64] whose autoscaling solution comprises two modules. The first module uses a generic autoscaling algorithm installed on the Google Kubernetes Engine (GKE) to determine the microservice resource needs. The algorithm adjusts the autoscaling of K8s according to the resource needs of the application. The second module uses Reinforcement Learning (RL) to learn and select autoscaling threshold values depending on resource demand and quality of service. The experimental findings demonstrate that the microservice response time may be improved by up to 20% when compared to the default autoscaling paradigm. Furthermore, RL can determine the threshold values without violating the response time SLA. This solution facilitates the adherence to QoS restrictions with little effort required from system users.

Beyond predicting workloads or learning thresholds, [67] proposes a complete replacement of the K8s horizontal pod autoscaler. The replacement is called *Ocean* and consists of a time-series replica predictor using one of two ML models: linear regression or Holt-Winters smoothing. Once the model is selected, it is fine-tuned using collected runtime data. The *Ocean* predictor is application-dependent, and its ML model must be re-trained for each application.

Another solution based on time series is *Predictkube* [71] which forecasts CPU and RPS time series using ML models. Its forecasts are then used for resource scaling based on a pre-defined threshold. Like [67], *Predictkube* must use a significant amount of application data for fine-tuning. The recommended time span for collected data is two weeks. *Predictkube* has been provided with an API for auto-scaling use in a variety of cloud environments, but the full documentation on its ML algorithms is yet to be published.

In June 2021, Amazon [63] started offering a predictive autoscaling plan that is applicable only to its EC2 instances. It considers the workloads of the deployed microservices and checks if the tasks submitted to the application need more capacity. If more capacity is needed, then an instance will be added to the cluster. This autoscaling plan has the following limitations: (1) the system load (that is, the number of tasks or requests from end users) is the target metric and only one resource metric (CPU or memory) can be used. The resource metric must be strongly correlated with the load metric; (2) the user must define the desired resource utilization, and scaling will be carried out to maintain this utilization; (3) the quality of the prediction model depends on the periodic pattern of the load. In addition, metrics for a minimum of 24 hours should be collected before starting the auto-scaling plan. However, it is recommended to wait 14 days for better accuracy; (4) finally, actions are based on rules and depend on the predefined threshold.

Al Qassem et al. [68] proposed a hybrid predictive two-state random forest autoscaler that predicts the expected CPU and memory usage based on current workload metrics. The predicted values are then used to adjust the resource pool vertically by allocating expected amount of needed hardware resources and horizontally by finding optimal number of microservice replicas. The proposed autoscaler unlocks significant efficiency and performance gains, demonstrated by a 90% boost in resource utilization coupled with a 95% improvement in end-to-end latency (measured by the 95^{th} percentile).

Xu et al. [69] tackled two key challenges in microservice management: workload prediction and auto-scaling decisions. They introduced a unit-based gradient recurrent algorithm for accurate workload predictions, thus enhancing the efficiency of scaling process. Building upon this, they proposed CoScal, a novel reinforcement learning approach, to optimize scaling decisions for diverse scenarios, aiming to minimize communication costs and delays. The proposed autoscaler is a self-adaptive bidirectional autoscaler. Their autoscaler combines vertical, horizontal, and brownout methods. The brownout method allows temporary deactivation of certain optional microservices to decrease resource consumption while maintaining the essential functionalities of microservice applications. CoScal has been intensively tested on two well-known microservice applications: Socks Shop application [76] and Stan's Robot Shop application [77]. The experiments illustrate that CoScal lowers response time by 19%-29% and connection time of services by 16%, in contrast to the cutting-edge scaling techniques for the Socks Shop application. It also enhances the number of successful transactions of Stan's Robot Shop application by 6%-10%. However, the main limitation of CoScal is that ML re-training to adapt to new application can be costly.

ChainsFormer [70] is another reinforcement-learning based bidirectional autoscaler that explores microservice interdependencies to identify critical paths (critical chains and nodes) in the microservice mesh. Due to the dynamic nature of microservice chains and workloads, ChainsFormer leverages

a lightweight machine learning model, named esDNN [78], to analyze the critical chains and predict the future workload. The reinforcement-learning model consider the predicted workload to scale the microservices horizontally and vertically. The proposed platform has been tested on the Train-Ticket application deployed on a Kubernetes-based cluster. The evaluation results showed that ChainsFormer, compared to three baselines: K8s horizontal autoscaler, Autopilot [79], and FIRM [59], can reduce response time by up to 26% and enhance the rate of processed requests by 8%.

While both CoScal [69] and ChainsFormer [70] are reinforcement learning-based frameworks utilizing machine learning for future workload prediction, there are notable distinctions between them. First, ChainsFormer has been deployed and tested at scale for microservice applications and is specifically designed for Kubernetes. However, CoScal has been deployed on Docker Swarm. Second, in contrast to ChainsFormer, CoScal does not employ chain analysis techniques for resource management within the microservice mesh. Finally, ChainsFormer utilizes the SARSA algorithm to update Q-values based on the current policy, enabling quicker convergence compared to CoScal, even though both incorporate reinforcement learning.

E. Discussion

Predictive models can be categorized into deterministic and probabilistic models. Most are deterministic models that use deterministic ML models and include point predictions without quantifying their uncertainty, avoiding the need for strict assumptions on the execution time distribution. This is the major limitation of most of the work reviewed in predicting the use of hardware resources of the underlying dynamic cloud workloads. Even though the deterministic supervised ML models produce adequate results, they are difficult to grow by adding new parameters (if necessary) to meet the changing features of cloud workloads. One way to address this challenge is to use progressive learning to improve the accuracy of predictive models [80], [81], [82], [83]. Progressive learning helps avoid retraining the model in all past data while maintaining prediction accuracy in past and current tasks [84]. Another approach is to use probabilistic models such as [85] which accounts for prediction uncertainty using Bayesian formulations.

It is important to note that all predictive platforms in this category scale microservices horizontally by predicting the number of replicas. None of the reviewed platforms predicts the amount of hardware resources needed for containerized microservices except [68]. Another important note is that latency is the main SLA metric that all platforms consider (see Table II), while none of them considers resource cost as an SLA metric. For batch workloads, throughput is a crucial SLA if the batch is the target workload.

In addition, the majority of predictive models in the literature focus on modeling and predicting request rates and/or resource utilization patterns, often overlooking workload distribution across microservices. The dynamic distribution of workloads among various microservices has the potential to trigger a cascading effect, significantly impacting the overall application performance. Consequently, addressing this challenge requires the development of new methods to simulate and standardize workload distribution within microservices for optimal load balancing and performance.

All in all, given the dynamic nature of cloud workloads, container orchestration systems leverage machine learning techniques to model and forecast multidimensional performance metrics [86]. These insights have the potential to enhance the accuracy of resource allocation decisions in complex scenarios, allowing for adaptability to changing workloads. In addition, ML algorithms have the potential to generate resource management decisions directly, rather than relying on heuristic methods. This can result in improved accuracy and reduced time overhead in large-scale systems.

TABLE III
COMPARISON OF HYBRID MICROSERVICE RESOURCE MANAGEMENT FRAMEWORKS

Components	RM	[59]	
	Schd	[87] [88]	
	AS	Vertical	[89]
		Horizontal	-
		Hybrid	-
	CL	-	
Hardware	CPU	All hybrid platforms	
	MEM	[89] [87], [59]	
	Network	[59]	
	I/O	[59]	
SLAs	Latency	[59], [87] [88]	
	Cost	[87]	
	Throughput	-	
	Reliability	-	
Workload	LC	[89] [87], [59] [88]	
	Batch	-	
	General	-	

VI. HYBRID FRAMEWORKS

We argue that a highly available data center with maximum utilization requires a robust hybrid scaling scheme that integrates both reactive and predictive components. In such a scheme, the reactive component would be appropriate during steady-state workloads, whereas the predictive component would be used more heavily during workload transients. As a result, the reactive component can rapidly respond to minor changes in resource demand, while the predictive component can pre-allocate resources in anticipation of greater demand. The hybrid frameworks discussed in this section are analyzed and compared in Table III.

In [89], an auto-scaling framework for microservices (ASFM) in the Infrastructure as a Service (IaaS) environment uses NN to predict workload along with a resource scaling optimization algorithm to scale resources and avoid SLA violations. The system consists of two main modules: (1) a predictive module that forecasts the workload using its HTTP request history; (2) a reactive module that finds the most efficient cloud computing package that meets the needed resources for the microservice application to avoid SLA violation. One aspect of [89] is that it explicitly addresses cost, since the system searches the cloud packages and selects the one with the lowest price for the needed resource. From the viewpoint of cost efficiency, the system works more as a cloud broker than as a microservice resource manager. In fact, the main objective of cloud brokers is to find the best cloud service

provider that best meets the user application requirements and resource specifications [90]. Another important aspect of the proposed system is that it continuously monitors the SLA status and notifies the scaling module (reactive module) whenever an SLA violation occurs to augment resources.

Bao et al. [87] also propose to combine a predictive component with a reactive one to minimize end-to-end latency and improve utilization. The predictive component predicts the processing time, which is the time interval between the submitted request and the result received from the microservice. The prediction of the processing time takes into account the time complexity of each microservice. The user request is modeled as an execution path in a directed acyclic graph, whose nodes are the function components of the microservice and whose edges form the dependency graph amongst the various functions. The predicted request delay is then calculated as the sum of execution times along the execution path. The reactive component uses the estimated performance and user's constraints to schedule microservices on the underlying hardware for minimum request delay under the user's constraints. This is achieved with a heuristic greedy recursive critical path algorithm [87].

FIRM [59] is a resource manager for latency-critical Web microservices based on a multilevel machine learning framework that manages resource sharing between microservices in a cluster. FIRM consists of two models: (1) a predictive support vector machine model to determine the microservice instances responsible for SLA latency violations by extracting the critical path; and (2) a reactive reinforcement learning model to reduce resource contention by adaptive reprovisioning decisions. FIRM outperforms the K8s autoscaler, as it reduces overall SLA violations by 16x compared to K8s.

The study by Xu et al. [88] addresses the challenge of resource allocation for large-scale microservice clusters while guaranteeing the performance of time-sensitive microservices. They introduced a hybrid scheduling algorithm that integrates both proactive and reactive scheduling decisions, taking into account various workloads. This algorithm is developed on top of Alibaba's microservice architecture, denoted as ASI. It involves employing diverse techniques like workload estimation, capability modeling, and resource allocation policies. The proposed resource provisioning algorithms represent an enhancement over Alibaba's current practices. These refinements improved the resource utilization by 10%–15% in Alibaba's clusters, while maintaining the latency QoS.

The status of hybrid frameworks is summarized in Table III. Note that all frameworks focus on LC workloads, but none of them supports a co-locator feature.

a) Discussion: Today, a few hybrid systems are available. This may be due to the difficulty of integrating reactive techniques with predictive ones and to the complexity that integration adds to the system. Another notable concern is the potential additional cost associated with managing two distinct systems concurrently. The integration complexity arises from the fact that the hybrid model should have the flexibility to apply the reactive, predictive, or optimal solution according to the workload under consideration. When the workload is periodic or quasiperiodic, the predictive approach is preferable, while the reactive approach is better suited for unpredictable and irregular workloads. One major challenge is the design and implementation of optimal adaptation policies specific to microservices. Such policies will be facilitated if the cloud computing industry adopts a certain level of microservice and container standardization.

Moreover, achieving a seamless integration of reactive and predictive approaches adds complexity to the overall architecture, demanding additional investments in expertise, tooling, and maintenance. This might offset potential cost savings from optimized resource utilization. For predictive systems, continuous collection of accurate data inputs and the implementation of sophisticated algorithms for future predictions are necessary, making it a potentially resource-intensive endeavor. Finally, careful calibration and tuning are essential for balancing both systems. However, setting improper thresholds or over-reliance on predictions can lead to inefficiencies and performance degradation.

In this sense, there is ample room for improvement, and significant effort is still needed to combine different reactive and predictive approaches to maximize resource utilization and minimize cost.

VII. MSA Management Framework Assessment

MSAs have gained widespread popularity among cloud software developers, as they enhance the efficiency and scalability of applications. However, container management is still an open problem [91].

In general, any container management model contains one or more of these four components: resource manager, scheduler, autoscaler, and task co-locator. The scheduler focuses mainly on distributing the loads among different microservice replicas to reduce response time and increase system throughput. The task co-locator classifies containers into two groups: LC and batch. The co-locator focuses on locating multiple LC and batch jobs on the same node to maximize resource utilization while meeting the SLA of each LC.

Most autoscalers scale containers either horizontally by changing the number of microservice replicas or vertically by adjusting the amount of resources assigned to the containers. Except for [32], none of the autoscalers examined considers the impact of scaling one microservice on other microservices. This shortcoming results in premature decisions, which in turn cause more scaling actions with potential QoS degradation.

The fourth component, the resource manager, focuses on resource allocation among microservices sharing the same host. This can be achieved with the following four actions: (1) Reduce: This action restricts the amount of resources allocated to each container; (2) Reuse: This action keeps the container in a 'warm' state, whereby its preprocessing steps are already executed, which enables it to run whenever needed. This means the containers are kept in the system in stopped state until they are needed again. This helps in assigning the previously allocated but currently unused resources to new container instances; (3) Recycle: This action removes the containers that have not been active for a long period. It completely removes the container and its associated resources

from the system; and (4) Refuse: This action stops spawning new containers if the host is overloaded.

Regardless of the components used, most existing solutions adopt centralized controllers to manage containers either on a single host or in a distributed fashion across the cloud nodes. The centralized design, however, has the disadvantage of being a single point of failure. There is a need for distributed controllers that can work together to manage containers deployed on different nodes in the cloud.

MSA-based applications pose significant challenges that should be considered when designing microservice frameworks. These challenges are:

1) **Prohibitively large action space**: Given the frequent changes in application behavior, resource management decisions must be made online. This requires the resource manager to efficiently explore a feasibility space that contains all possible resource assignments per microservice [61]. An exhaustive evaluation of all actions under various workloads is time consuming and compute intensive. As a result, resource management requires efficient action space pruning methods and computational techniques with good generalization capabilities.
2) **Dependencies between microservices**: Microservice-based systems consist of dependent microservices that exchange data and dynamically interact with each other. This dynamic interaction not only affects resource management but also poses major challenges in assessing system behavior and detecting crucial resource bottlenecks.
3) **Lack of end-to-end visibility**: For service providers, user applications are "black boxes" that do not expose the business activities or the specific business demands of the hosted applications. Therefore, service providers rely on external application metrics to dynamically scale services. However, the limited information on the application phases and workloads increases the uncertainty about upcoming demands and resource deployment.
4) **Diversity of cloud workloads**: Supporting different classes of applications (AI training, Web services, social networks, data mining, etc.) makes it difficult to deal with bottlenecks at the hardware-software interface. What makes matters even worse is that users and vendors have different perceptions of QoS metrics. The user's QoS metrics, which are referred to as quality of experience (QoE), focus on the user's experience and include factors such as perceived performance. This factor is directly related to the latency SLA, which is also an important QoS metric. However, QoS and QoE have different perspectives. QoS measures latency from a technical standpoint, such as round-trip time or network delay. On the other hand, QoE evaluates latency from the end user's standpoint and considers subjective factors such as the user's perception of latency, their level of satisfaction with the system's performance, and the impact of latency on their overall experience.
5) **Microservices co-location**: Co-locating distinct microservices can result in cost reduction, but it can also cause significant fluctuations in resource utilization due to workload variance [88]. Failure to allocate resources promptly in such scenarios can translate into excessive latency, underscoring the need for robust resource management strategies within microservices architectures.
6) **Large scale microservice clusters**: The scaling up of microservice architectures to large-scale clusters in industry and production environment, highlights the limitations of conventional resource provisioning methods validated in smaller, research-oriented settings [88]. Alibaba, for instance, has transitioned its services to cloud-native clusters, heavily employing microservices-based technologies [92]. By 2021, the scale of Alibaba's cloud-native clusters had reached 10 million cores, and its infrastructure success in handling 0.58 million transactions per second during China's Double 11 Shopping Festival [88]. This mismatch between established practices and demands of real-world scenarios necessitates the development of novel resource provisioning strategies specifically tailored for large-scale cloud-native ecosystems.

These challenges indicate that experimental resource managers, such as rule-based and/or statistical autoscalers, are vulnerable to unpredictable requests and/or resource inefficiency. ML is a promising approach to overcome these challenges [61]. Indeed, ML models promise to estimate end-to-end latency and the likelihood of a QoS violation for a resource configuration even when service execution times are non-stationary or even non-parametric. The resource manager then uses these forecasts to maximize resource utilization while satisfying QoS. This necessitates the collection of historical resource usage data for statistical analysis to build effective prediction models and develop workload-specific scheduling algorithms that ensure optimal resource allocation.

Tables I, II, III and V summarize the resource management frameworks reviewed in this paper. Most of them target latency-critical workloads that are mainly interactive Web-based services. One model [34] targets CPU-intensive High Performance Computing (HPC) applications. As shown in Table V, the apple-to-apple evaluation of these models is hampered by the lack of common benchmarks. In addition, several of them have been compared with their own baseline cases, thus complicating the comparisons amongst their various attributes. For example, only three models [51], [59], [65] have compared their work with K8s, a widely used container orchestration system in the industry. Therefore, we have provided in Table V the platforms to which the proposed solution was compared and the evaluation results of such a comparison (the last two columns). This will help researchers make a decision on the best platform to meet their system requirements. In other words, the solutions are workload-dependent; hence, no solution is better than another. For example, some systems are designed for latency-critical task allocation like [47], [48], [49], while others are designed to allocate resources for compute-intensive tasks like [66].

Moreover, all models except [52] have considered CPU utilization as the main evaluation metric. They have calculated

CPU utilization as the ratio of CPU consumption to the previously allocated CPU. This is because most of the models target LC jobs, which typically require significant compute resources. Furthermore, CPU is a major bottleneck for Web applications [93], and according to [94], CPU utilization of compute-intensive workloads increases with the number of requests and therefore has a monotonic dependence on the request rate. The response time, in turn, depends monotonically on CPU utilization, with the dependence becoming exponential as the request queue becomes full.

According to the analysis of 15 TB of raw data of Alibaba's large-scale microservices run-time information performed in [88], the resource requirements of each microservice instance can fluctuate depending on the volume of processed requests. To reduce the cost and maximize resource utilization, Alibaba co-locates the microservices. The co-location has shown distinct utilization patterns for CPU and network compared to the relatively stable memory usage. This suggests that, for large-scale deployed business microservices, CPU and network are the primary resource bottlenecks in Alibaba's deployment, while out-of-memory issues occur less frequently. In addition, another study [95] on workload analysis of large-scale deployments of microservices at Alibaba clusters has demonstrated that large-scale microservices are more sensitive to CPU interference compared to memory interference. This finding simplifies scheduling by shifting the focus to balanced CPU utilization across hosts, potentially leading to more efficient resource management and smoother scheduling workflows.

However, if only the CPU is used to manage the workload, management will fail with memory-intensive workloads. In the latter case, memory becomes the bottleneck, with cache misses and page faults increasing the response time and resulting in SLA violations. Consequently, several of the reviewed frameworks [45], [51], [59], [62], [65] include both CPU and memory requirements to manage microservices and their workloads. A crucial phase in microservice resource allocation is during the allocation of containers and tasks, when the demand for CPU and memory is at its peak.

While the majority of the reviewed approaches have made significant contributions to microservice resource provisioning, they are primarily based on small-scale practices, potentially limiting their applicability to real-world production environments. One recent work [88] addresses the challenges of large-scale microservice clusters by introducing a hybrid resource provisioning algorithm. This algorithm optimizes Alibaba's approach while ensuring the maintenance of latency QoS.

With respect to SLAs, the reviewed models have focused mainly on the latency of critical jobs and the throughput of batch jobs. Further work may be needed in this area where co-location schedulers of time-critical and batch jobs are more aware of SLA constraints.

Another important observation from Table I is that most reactive frameworks feed into optimization frameworks to find the most appropriate resource configuration. The general mathematical formulation of the microservice resource optimization problem can be summarized as follows:

1) Define the system components: The system usually consists of applications running in one or more containers, physical machines and/or virtual machines, and a network.
2) Define the microservice mesh: The microservices are modeled as a directed graph, in which each edge is represented by the pair $(\mu S_i, \mu S_j)_{prod/cons}$, where the results of the producer microservice μS_i are consumed by the consumer microservice μS_j
3) Define the microservice state: Each microservice i is represented as a tuple: $(Scale_i, res_i, thresh_i, fail_i, \ldots)$ whose components are the optimization arguments, with res_i being the most important component, since it represents the resource requirements of the microservice. All reviewed models have included CPU requirements in res_i, but memory and network requirements have been introduced in the most recent models in an effort to improve resource utilization and reduce SLA latency. When microservice reliability and availability are included in optimization objectives, the microservice failure rate $fail_i$ becomes an important optimization argument. Each microservice is executed in one or more containers, with the number of running containers depending on the scaling level $Scale_i$, which is an essential metric used to calculate the total resource consumption of a microservice. Finally, $thresh_i$ is the CPU threshold for the microservice and is used to find the desired number of containers to avoid performance degradation. For example, [1] uses the threshold distance as the objective function that should be minimized to determine the value of $Scale_i$. The threshold distance is defined as:

$$Thresh = \sum_i \left| \frac{Requests(\mu S_i) * res_i}{Scale_i} - thresh_i \right|$$

To find the desired number of containers, the following equation is used:

$$Scale_i = \frac{Total\_CPU\_Usage(\mu S_i)}{thresh_i}$$

4) In multinode systems, the physical/virtual machines are also characterized by pairs to identify their resource capacity and failure rate $(Cap_i, fail_i)$.
5) The most important step is to define the objective function. Reducing the overall cost of deploying containers on one or more nodes is a standard objective function in all models and is achieved by maximizing resource utilization while still meeting SLAs. Another standard objective function is to reduce the response time by reducing the communication latency between microservices. Such a reduction will, of course, help to meet the system SLA.
6) A related step is to define the constraints under which the objective function is optimized. Some of the common constraints are listed below.
   a) The amount of resources allocated to each container should be greater than or equal to res_i.

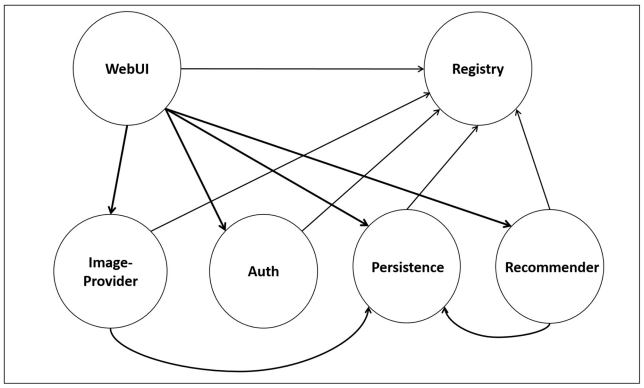

Fig. 3. Architecture of the Teastore microservice [96].

b) Only one container instance from the same microservice is running on any physical node. This helps to reduce resource conflicts between containers of the same microservice.
c) The total amount of resources allocated to microservices running on a physical node or VM must not exceed its capacity.

Although optimization-based solutions reported very good results, the proposed solutions must target one specific microservice application. This is due to the limited availability of the necessary information to construct the optimization problem (like the microservice mesh and their producer/consumer dependencies). Therefore, it is difficult to design an optimization problem that is application-independent. This challenge for optimization methods is, in fact, an opportunity for machine learning-based methods where supervised and semi-supervised models can be used to compensate for the lack of available information [10], [68].

VIII. MSA Efficiency

Despite numerous proposals for efficient management of cloud infrastructure resources, it remains a significant challenge for the leading cloud service providers (Google, Amazon, and Alibaba). One main reason is the limited predictability of the underlying workload. Google [3] and Alibaba [97] have recently performed trace analysis to provide researchers with first-hand information on their real cloud workloads. This information is necessary to design better scheduling and resource management systems.

Understanding the properties of application workloads provides information on their resource utilization and, therefore, can be used to improve the performance of software and hardware configurations [98]. Additionally, understanding the dynamic nature of cloud workloads may help maintain the accuracy of resource allocation machine learning models over time [99]. Furthermore, understanding microservice workloads provides a direct way to improve resource utilization in data centers. This can be achieved by co-locating LC workloads (such as Web search and social networks that require a minimum amount of resources to avoid SLA violations) and batch/HPC workloads (such as scientific computing and weather forecasting that can withstand long wait times)

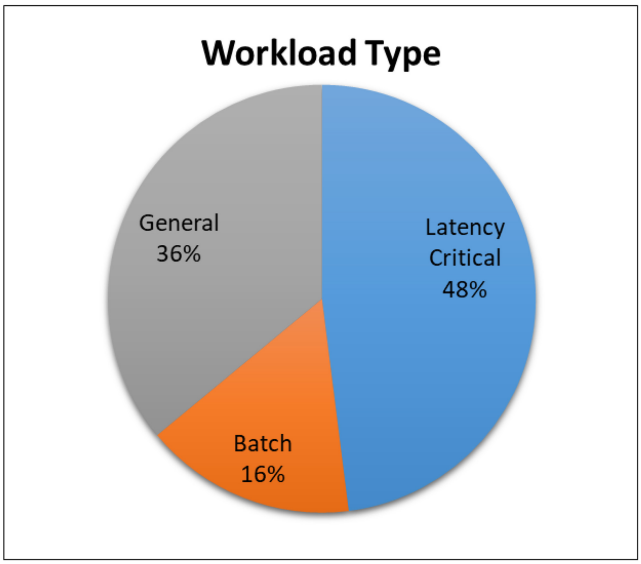

Fig. 4. Pie chart of the major microservice workload types.

on the same hardware [3]. Several management policies in the literature take into account the type of workload as a policy parameter [48], [62], while others target specific types of workload [47], [65]. In addition, several of them focus on co-locating different workloads on the same host to maximize resource utilization, for example, Heracles [47], PARTIES [48], and CLITE [49].

Fig. 4 shows the percentage of different workloads that have been targeted by the platforms analyzed in this study. Almost half (48%) of all studies have focused on the LC workload. On the other hand, only 16% of the platforms target batch workloads that are compute intensive and run in the background. The remaining studies generalized their solutions to any type of workload.

In general, clusters contain a large number of parallel workflows that run independently. These workflows differ in their attributes, such as execution time, data volume, and network latency, but all share the same cluster resources. As a result, calculating the overall resource usage for the entire workflow is complicated. Node performance is calculated at the start of workflow execution for node-based static load balancing policies. Therefore, all tasks in that workflow will be performed on the same node to which they have been allocated, resulting in unbalanced resource allocation. In contrast, breaking down resource allocation to the granularity of a container, each responsible for a single application or service, will help to distribute the workload between nodes in a more balanced manner.

Several proposed solutions [1], [7], [57] represent the workflow of microservice applications as a directed acyclic graph (DAG), as requests in workflows usually span numerous microservices. A DAG can help to describe data dependencies, estimate request execution and completion times, and efficiently allocate containers in a cluster. The DAG diagram in Fig. 3 shows the interactions and dependencies between the microservices of the Teastore application [96] that was used by [32] to evaluate their resource management

policy. This is one of the main limitations of the reviewed literature. Typically, information about the DAG of the running application is not available to cloud providers, who are responsible for improving resource utilization and power management. Furthermore, cloud providers cannot improve execution efficiency if the microservice application has a long critical path(s) or if the microservices' algorithms are slow. This is why some proposed solutions ([1], [32]) are designed for specific microservice applications and cannot be used for others.

One last crucial point that was not considered in these reviewed articles is security. This should be addressed since the sharing of resources among different containers is similar to the sharing of resources among tenants in the cloud [100]. Therefore, microservice-based management systems must provide security for memory access, network access, and CPU computing to avoid data or app tampering that could crash the system or damage hardware resources. Thus, a security layer between containers is needed to avoid attacks from malicious users. According to [101], Dockerhub has discovered that several of its images were being exploited maliciously to mine for cryptocurrencies. Attackers have utilized the Docker image to implant cryptojacking malware, since it can be accessed without authentication.

A. Cost Efficiency

Interestingly, data centers are approaching their physical and financial limitations in terms of capacity, power consumption, resource usage, and operational costs [102]. Therefore, it is not always feasible to scale horizontally and offer more resources as a buffer against SLA violations. This solution increases both the cost of data centers and power consumption due to the addition of more machines to the resource pool. Cloud clusters, on the other hand, may suffer from resource under-utilization. During off-peak hours, tenants are usually oversupplied with resources. These unused resources can be reallocated to save energy and can be easily transferred to another tenant if there is an immediate need. data centers are also heterogeneous in nature, which means that machines run at various clock frequencies and have varying memory limits. Enhancing the efficiency of resource usage on each machine while reducing the number of machines used presents another approach to reducing operating cost. In addition, machines run the risk of being overloaded if their specifications are not considered. For example, exceeding memory limits causes computers to swap to disk, resulting in high latency and overall performance degradation. Therefore, efficient microservice resource management should be a top priority for data centers, as it can lead to significant cost savings.

All reviewed solutions have designed their models with cost efficiency as the target, but quantitative breakdowns of the cost reductions achieved have not been provided. For example, the systems described in [41], [89] can advantageously operate as cloud brokers that aim to reduce the cost of deploying MSA systems in the cloud by selecting the cloud package with the lowest cost while meeting the minimum requirements of MSA. Unfortunately, no cost savings evaluation results have been provided.

B. Energy Efficiency

The resource solutions reviewed in this article aim to improve resource utilization for MSA systems in the cloud, which, in turn, contributes to reducing their data center energy footprint. Various algorithms have been proposed to implement resource allocation strategies to minimize energy consumption. For example, Sinan's scheduler [61] has two phases: core allocation and power management. In the first phase, the scheduler reduces the number of cores until no further reduction is possible. The scheduler then enters the power management phase and gradually decreases frequency. The work in [103] suggests a greedy container placement strategy that assigns containers to the most energy-efficient servers. The greedy algorithm was compared to random task allocation and the least-allocated-server-first scheduling scheme using Google trace data [3]. The simulation results showed that the proposed greedy scheme can significantly reduce the power consumption compared to the other two approaches. Heracles [47] also showed a significant improvement in energy efficiency (2.3X to 2.4X gain in energy efficiency). The self-adaptive approach proposed in [104] leverages container technology to reduce power consumption in data centers. It is based on a reactive model designed to minimize brown energy consumption and maximize renewable energy consumption while ensuring compliance with SLAs. The model targets both interactive and batch workloads. The proposed method successfully achieves a 21% reduction in non-renewable energy consumption and a 10% increase in renewable energy utilization.

Other systems utilize the heterogeneity feature of the cloud for energy savings. For example, the microservice design technique proposed in [105] decomposes the applications into a set of microservices. Then it schedules these microservices on the appropriate compute node. The purpose is to take advantage of heterogeneous hardware to maximize energy efficiency.

A consolidation of containers and virtual machines in the cloud is proposed to run cloud services in [106]. The presented solution provides a new cloud resource management strategy based on a multi-criteria decision-making method that utilizes a combined virtual machine and container migration methodology at the same time. The results of the simulations exhibit significant reductions in energy consumption, SLA violation, and migration number compared to state-of-the-art algorithms.

On the other hand, predictive frameworks that utilize ML to manage microservice applications, especially in large-scale deployment scenarios, lack in-depth cost analysis [86]. While ML technologies have demonstrated significant efficacy in addressing various challenges within microservice management frameworks, their computational demands can hinder performance or increase energy consumption, particularly in online decision-making or energy-efficient scenarios. Therefore, ML-based solutions require more comprehensive cost analysis and an effective strategy to tackle this issue.

TABLE IV
COMPARISON OF THE THREE RESOURCE MANAGEMENT FRAMEWORKS

	Reactive	Predictive	Hybrid
Popularity	Widely adopted approach (57.69%) Cutting-edge technology (e.g. K8s[38])	More popular than hybrid (30.7%)	Least adopted approach (11.5%) [only 3 systems]
Implementation	Easiest to implement as a rule-based solution; Optimization difficulty is considered moderate	Challenging implementation due to learning phase	Most complex due to integration.
Algorithm	Rule-based Optimization technique	ML	Optimization & ML
Limitation	Large delays between observations and responses Non-optimal resource configurations Optimization solution is application-dependent.	Lack of training data	Integrating reactive & predictive models
Target Application	Web-based HPC apps (1 system) Latency-critical (task co-locator)	Web-based ML/AI apps	Web-based

One promising approach is optimizing the trade-off between offline and online training. For instance, in the Reinforcement Learning approach, the knowledge pool can accumulate a greater number of well-trained decisions based on historical data, allowing reat-time decisions to rely more on this pool than on online training [86].

C. Model Comparison

As discussed above, the different microservice management systems can be categorized into three categories. Table IV summarizes the features and limitations of the three types of resource management frameworks. The main goal of these models is to maximize the utilization of shared hardware resources, which is a challenging task due to the dependency between microservices that increases the complexity of the cluster manager. However, a change in the application workload over time could require more resources than those assigned to microservices. Therefore, over-provisioning of resources would be insufficient to avoid SLA violations. To maximize resource utilization, there is a need for a flexible platform that can dynamically scale microservices horizontally (i.e., scaling the number of containers for a microservice) or vertically (i.e., increasing/reducing the amount of resources available to a microservice container).

IX. CONCLUSION

Microservices are advantageous when a degree of flexibility is needed in a cloud computing framework. They also facilitate software migration and development. However, the use of microservices introduces many complications that will require a significant amount of effort to overcome.

To date, major cloud service providers, like Amazon, IBM, and Google, have extended their cloud services to provide container-as-a-service (CaaS) to automate container deployment on their offered IaaS. Amazon ECs [107], IBM Bluemix [108], and Google Container Engine [109] are some examples of CaaS. Therefore, container allocation and resource management are key factors for cloud providers as they influence system performance and resource consumption. Unfortunately, the problem of efficiently handling computational resources for containers remains unsolved. Applications that share the same containers may face severe resource conflicts. In addition, resources are dynamically reallocated between containers, making efficient job scheduling, load balancing, and resource distribution crucial to maintaining the scalability and high availability of a virtualized computing environment. Finally, management systems should provide security for sharing hardware resources to avoid data tampering or running malicious containers.

Our view is that the ideal microservice management platform should consider both application virtualization and hardware virtualization, as they give the flexibility to find the optimal set of policies for containerized MSA. Furthermore, such an ideal platform should support two levels of security: the container level and the microservice level. The container level is needed to securely manage shared hardware resources among different containers. The microservice level is needed to secure a service-to-service communication layer that controls and manages messaging between the microservices of a given mesh. Finally, the ideal platform must be green in that its energy consumption must be minimized for a given workload by judicious allocation of hardware resources. Such resources should be expanded to include not just CPUs and RAMs in support of microservice replicas, but also GPUs, TPUs, and FPGAs in support of specialized workloads such as AI, e-Commerce, or IoT.

In summary, we have reviewed the latest work and solutions for efficient microservice management platforms. We have classified existing resource management frameworks into three categories: reactive, predictive, and hybrid. We have discussed the advantages and limitations of each of them. Regardless of the category of resource management framework, we have found that the most efficient approach to allocating and distributing HW resources across a microservice ecosystem is by abstracting hardware resources (CPU, memory, storage, etc.) using abstraction technologies such as Apache Mesos [36]. This enables dynamic allocation of resources in large-scale distributed systems. Additionally, microservices should be classified according to their importance and value to the overall business. For example, business-critical services should receive a larger share of resources. It is also important to identify the resource requirements of each microservice before allocating the HW resources that each service needs to run properly. Identifying these is essential to run a scalable service and determine the overall performance and scalability of microservices. Furthermore, the current body of research on microservice resource provisioning, predominantly tested

TABLE V
EVALUATION OF MICROSERVICE RESOURCE MANAGEMENT FRAMEWORKS. THE PLATFORMS ARE GROUPED ACCORDING TO THEIR CATEGORY BY A TWO-LINE SEPARATOR BETWEEN THE GROUPS. THE FIRST GROUP REPRESENTS REACTIVE PLATFORMS. THE SECOND GROUP REPRESENTS PREDICTIVE PLATFORMS. THE THIRD GROUP REPRESENTS HYBRID PLATFORMS

Platform	Dataset	Comparison Benchmark	Benchmarking Assessment
[35][a.]	LINPACK[110]	Baseline design	Runtime improved by 4X
[41]	Case study	State-of-the-art deployment strategies (Spread, Random, and BinPack)	Performance similar to BinPack
[1]	Socks Shop [76]	K8s policies	Better resource utilization Lower energy consumption
[42]	Socks Shop [76]	[1]	27% cost reduction in multi-objective criteria compared to [1]
[43]	Own Data	2 baseline VM autoscalers	20-28% cost reduction
ECVD [44]	Own Data	Greedy First-Fit Round Robin (RR)	ECVD smilar to Greedy ECVD shorter deployment time RR most costly & longest deployment time
[45]	NGINX server	Round Robin (RR)	Faster servers by approx 15%
Heracles [47]	Google production latency-critical workloads Google brain workload	baseline	90% utilization with no SLA violations
PARTIES [48]	LC: Memcached, Xapian, NGINX, Moses, MongoDB, & Sphinx Batch: Own Data	Heracles [47] Unmanaged controller Oracle	61% better utilization than Heracles Higher throughput
CLITE[49][a.]	LC: Tailbench benchmark suite [112] Batch: PARSEC benchmark suite [111]	Heracles [47] PARTIES [48] Oracle	Improves latency-critical performance by $> 15\%$ with no SLA violations
Reactive HyScale[51]	Own Data	K8s	59% lower response time with no failure
Atom[33]	Sock Shop microservices-demo.github.io	2 baselines (rule-based autoscaler): vertical & horizontal	30%-37% higher throughput
Erms[46]	DeathStarBench[56]	GrandSLAm [57], Rhythm [58], and Firm [59]	Reduces latency SLA violation probability by 5× Reduces resource usage by 1.6×
CTCL [65]	Alibaba 2018 cluster traces	K8s Baseline: Alibaba Trace	Improves CPU utilization by 62% and memory utilization by 12%
EMARS[52]	OpenLambda latest Code	Own Data	Qualitative: Assigning memory limits helps in efficient resource management
RScale[7]	NSF Cloud's Chameleon Benchmark: Robot Shop	Comparing GP uncertainty with that of NN	NN: 4X more overhead; target CPU utilization: missed and target SLA latency: missed
GrandSLAm[57]	AI & ML microservices	Baseline [FIFO+no sharing uServices]	Throughput increase by up to 3X without SLA violations
Sinan [61]	Social Network from DeathStarBench [56]	CNN compared to MLP & LSTM Autoscaling policies: AS_Opt & AS_Cons	CNN: lowest RMSE on tail latency & smallest model size Sinan: no SLA violations, resources saving
FI system [62]	Own Data	Round Robin	Container selection: FI is better, especially at high workloads
[32]	Teastore [96]	K8s horizontal autoscaler	Higher throughput, lower response time fewer actions to maintain QoS
Predictive HyScale[31]	fastStorage, GWA-T-12[113]	Tested 4 NNs models	LN-LSTM has lowest RMSE on CPU utilization
Proactive RF[68]	fastStorage[113]	Predictive HyScale[31]	RF has lower RMSE
CoScal [69]	Alibaba Trace Analysis [114]	DoScal: Docker Swarm Vertical AS K8s Horizontal AS HyScals [51]	CoScal outperformed all baselines and improve resource utilization while maintaining latency SLA.
ChainsFormer [70]	Alibaba Trace Analysis [114]	K8s Horizontal AS Autopilot [79] Firm[59]	CoScal outperformed all baselines reduce response time by 26% & enhance processed request rate by 8%.
[87][a.]	ACM Air [115]	Greedy A* [116], BHEFT [117]	20% latency improvement compared with Greedy A* and 35% compared with BHEFT
FIRM[59][a.]	DethStarBench Train-Ticket	K8s Additive increase multiplicative decrease (AIMD) based methods	Reduces SLA violation by 16x & latency by 10x Increases Average CPU utilization by up to 33%
ASFM[89]	FIFA World Cup 98	ARIMA model [118]	Qualitative RMSE improvement on number of HTTP requests and forecasting horizon
Ali-pro[88]	E-business App on Alibab cluster	Alibaba's resource provisioning solution Kube-pro [119] Optimal-pro [120]	Ali-pro improves resource utilization by 10%-15% compared to baselines

on small-scale studies, exhibits a critical gap in effectively translating its findings to the realities of large-scale production environments. This necessitates the development of robust and scalable resource provisioning methodologies tailored for real-world application. Bridging the gap between research and real-world production environments necessitates further investigation into scalable and generalizable solutions. Finally, we have sketched a blueprint for future improvements to microservice resource management frameworks in the cloud.

REFERENCES

[1] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *J. Grid Comput.*, vol. 16, no. 1, pp. 113–135, 2018.

[2] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *Proc. 43rd ACM/IEEE Annu. Int. Symp. Comput. Architecture (ISCA)*, 2016, pp. 456–468.

[3] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–13.

[4] H. Liu, "A measurement study of server utilization in public clouds," in *Proc. 9th IEEE Int. Conf. Dependable, Auton. Secure Comput.*, 2011, pp. 435–442.

[5] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of Alibaba data center traces," in *Proc. 27th IEEE/ACM Int. Symp. Qual. Service (IWQoS)*, 2019, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3326285.3329074

[6] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries," in *Proc. Symp. Cloud Comput.*, 2017, pp. 611–623.

[7] P. Kang and P. Lama, "Robust resource scaling of Containerized Microservices with probabilistic machine learning," in *Proc. 13th IEEE/ACM Int. Conf. Utility Cloud Comput. (UCC)*, 2020, pp. 122–131.

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Softw.*, vol. 33, no. 3, pp. 42–52, May/Jun. 2016.

[9] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 81–88, Sep./Oct. 2016.

[10] B. D. Martino, A. Esposito, and E. Damiani, "Towards AI-powered multiple cloud management," *IEEE Internet Comput.*, vol. 23, no. 1, pp. 64–71, Jan./Feb. 2019.

[11] W. Hussain, F. K. Hussain, O. K. Hussain, E. Damiani, and E. Chang, "Formulating and managing viable SLAs in cloud computing from a small to medium service provider's viewpoint: A state-of-the-art review," *Inf. Syst.*, vol. 71, pp. 240–259, Nov. 2017.

[12] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in DevOps," *J. Syst. Softw.*, vol. 170, Dec. 2020, Art. no. 110798.

[13] M. Waseem, P. Liang, G. Márquez, and A. Di Salle, "Testing Microservices architecture-based applications: A systematic mapping study," in *Proc. 27th Asia-Pac. Softw. Eng. Conf. (APSEC)*, 2020, pp. 119–128.

[14] G. Marquez, F. Osses, and H. Astudillo, "Review of architectural patterns and tactics for microservices in academic and industrial literature," *IEEE Latin America Trans.*, vol. 16, no. 9, pp. 2321–2327, Sep. 2018.

[15] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018.

[16] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, "Design principles, architectural smells and refactorings for microservices: A multivocal review," *SICS Softw. Intensive Cyber Phys. Syst.*, vol. 35, no. 1, pp. 3–15, 2020.

[17] D. Taibi, V. Lenarduzzi, and C. Pahl, "Continuous architecting with microservices and devops: A systematic mapping study," in *Proc. Int. Conf. Cloud Comput. Services Sci.*, 2018, pp. 126–151.

[18] C. Pahl, P. Jamshidi, and O. Zimmermann, "Microservices and containers: Architectural patterns for cloud and edge," in *Proc. Softw. Eng.*, 2020, pp. 115–116.

[19] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *Proc. 6th Int. Conf. Cloud Comput. Services Sci.*, 2016, pp. 137–146.

[20] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, Sep./Oct. 2017.

[21] E. Casalicchio, "Container orchestration: A survey," in *Systems Modeling: Methodologies and Tools*. Berlin, Germany: Springer Int. Publ., 2019, pp. 221–235.

[22] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 7, pp. 3934–3947, 2022.

[23] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Jul.–Sep. 2019.

[24] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *Proc. 38th Int. Conf. Chilean Comput. Sci. Soc. (SCCC)*, 2019, pp. 1–7.

[25] "Use containers to build, share and run your applications." Docker. 2023. Accessed: Mar. 13, 2023. [Online]. Available: https://www.docker.com/resources/what-container/

[26] (IBM, Armonk, NY, USA). *What Are Containers?* (2023). Accessed: Mar. 13, 2023. [Online]. Available: https://www.ibm.com/ae-en/topics/containers

[27] L.-J. Jin, V. Machiraju, and A. Sahai, "Analysis on service level agreement of Web services," *HP June*, vol. 19, pp. 1–13, Jun. 2002.

[28] K. Saravanan and M. Rajaram, "An exploratory study of cloud service level agreements-state of the art review." *KSII Trans. Internet Inf. Syst.*, vol. 9, no. 3, pp. 843–871, 2015.

[29] D. Serrano et al., "Towards QoS-oriented SLA guarantees for online cloud services," in *Proc. 13th IEEE/ACM Int. Symp. Cluster, Cloud, Grid Comput.*, 2013, pp. 50–57.

[30] M. Gribaudo, M. Iacono, and D. Manini, "Performance evaluation of massively distributed microservices based applications," in *Proc. 31st Eur. Conf. Model. Simul. (ECMS)*, 2017, pp. 598–604.

[31] J. P. Wong, "Hyscale: Hybrid scaling of dockerized microservices architectures," Ph.D. dissertation, Dept. Electr. Comput. Eng., Univ. Toronto, Toronto, ON, Canada, 2019.

[32] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian, "A holistic machine learning-based autoscaling approach for microservice applications," in *Proc. 11th Int. Conf. Cloud Comput. Services Sci.*, 2021, pp. 190–198.

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

[34] J. Monsalve, A. Landwehr, and M. Taufer, "Dynamic CPU resource allocation in containerized cloud environments," in *Proc. IEEE Int. Conf. Clust. Comput.*, 2015, pp. 535–536.

[35] S. Herbein et al., "Resource management for running HPC applications in container clouds," in *Proc. Int. Conf. High Perform. Comput.*, 2016, pp. 261–278.

[36] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implement.*, 2011, pp. 295–308.

[37] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implement.*, 2011, pp. 24–24.

[38] "Kubernetes." 2020. Accessed: Feb. 16, 2020. [Online]. Available: https://kubernetes.io/

[39] D. Vohra, "Scheduling pods on nodes," in *Kubernetes Management Design Patterns*. Berkeley, CA, USA: Apress, 2017, pp. 199–236.

[40] "Swarm mode overview." Docker. 2021. Accessed: Feb. 16, 2021. [Online]. Available: https://docs.docker.com/engine/swarm/

[41] N. D. Keni and A. Kak, "Adaptive containerization for microservices in distributed cloud systems," in *Proc. 17th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, 2020, pp. 1–6.

[42] K. N. Vhatkar and G. P. Bhole, "Particle swarm optimisation with grey wolf optimisation for optimal container resource allocation in cloud," *IET Netw.*, vol. 9, no. 4, pp. 189–199, 2020.

[43] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, "Four-fold auto-scaling on a contemporary deployment platform using docker containers," in *Proc. Int. Conf. Service Oriented Comput.*, 2015, pp. 316–323.

[44] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. Companion*, 2017, pp. 5–10.

[45] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Proc. 9th IEEE Int. Conf. Knowl. Smart Technol. (KST)*, 2017, pp. 254–259.

[46] S. Luo et al., "ERMS: Efficient resource management for shared microservices with SLA guarantees," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022, pp. 62–77. [Online]. Available: https://doi.org/10.1145/3567955.3567964

[47] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2015, pp. 450–462.

[48] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 107–120.

[49] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2020, pp. 193–206.

[50] A. F. Baarzi and G. Kesidis, "SHOWAR: Right-sizing and efficient scheduling of microservices," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 427–441.

[51] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "HyScale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2019, pp. 80–90.

[52] A. Saha and S. Jindal, "EMARS: Efficient management and allocation of resources in serverless," in *Proc. 11th IEEE Int. Conf. Cloud Comput. (CLOUD)*, 2018, pp. 827–830.

[53] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: A case study in multi-cloud microservices-based applications," *J. Supercomput.*, vol. 74, no. 7, pp. 2956–2983, 2018.

[54] A. Scionti et al., "The green computing continuum: The OPERA perspective," in *Hardware Accelerators in Data Centers*. Berlin, Germany: Springer Int. Publ., 2019, pp. 57–86.

[55] "swarmkit," Docker. 2021. Accessed: Mar. 22, 2021. [Online]. Available: https://github.com/docker/swarmkit

[56] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 3–18.

[57] R. S. Kannan et al., "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.

[58] L. Zhao et al., "Rhythm: Component-distinguishable workload deployment in datacenters," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–17.

[59] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. 14th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2020, pp. 805–825.

[60] C. A. Ardagna, E. Damiani, F. Frati, D. Rebeccani, and M. Ughetti, "Scalability patterns for platform-as-a-service," in *Proc. 5th IEEE Int. Conf. Cloud Comput.*, 2012, pp. 718–725.

[61] Y. Zhang, W. Hua, Z. Zhou, E. Suh, and C. Delimitrou, "Sinan: Data-driven resource management for interactive microservices," *ML Comput. Archit. Syst.*, 2020, pp. 1–6.

[62] Y. Tao, X. Wang, X. Xu, and Y. Chen, "Dynamic resource allocation algorithm for container-based service computing," in *Proc. 13th IEEE Int. Symp. Auton. Decent. Syst. (ISADS)*, 2017, pp. 61–67.

[63] (Amazon Web Services, Seattle, WA, USA). *AWS Auto Scaling*. (2021). Accessed: May 5, 2022. [Online]. Available: https://docs.aws.amazon.com/autoscaling/plans/userguide/how-it-works.html

[64] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021.

[65] Z. Zhong, J. He, M. A. Rodriguez, S. Erfani, R. Kotagiri, and R. Buyya, "Heterogeneous task co-location in Containerized cloud computing environments," in *Proc. 23rd IEEE Int. Symp. Real-Time Distrib. Comput. (ISORC)*, 2020, pp. 79–88.

[66] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1448–1460, May/Jun. 2022.

[67] J. Thompson. "Predictive horizontal pod autoscaler." 2021. [Online]. Available: https://github.com/jthomperoo/predictive-horizontal-pod-autoscaler

[68] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. A. M. Elfadel, "Proactive random-forest autoscaler for microservice resource allocation," *IEEE Access*, vol. 11, pp. 2570–2585, 2023.

[69] M. Xu et al., "CoScal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 4, pp. 3995–4009, Dec. 2022.

[70] C. Song et al., "ChainsFormer: A chain latency-aware resource provisioning approach for microservices cluster," in *Service-Oriented Computing*, F. Monti, S. Rinderle-Ma, A. Ruiz Cortés, Z. Zheng, and M. Mecella, Eds. Cham, Switzerland: Springer Nat., 2023, pp. 197–211.

[71] KEDA. "Predictkube." 2020. [Online]. Available: https://predictkube.com/

[72] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2018, pp. 427–444.

[73] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers," *Softw., Pract. Exp.*, vol. 47, no. 4, pp. 505–521, 2017.

[74] (Alibaba Group E-commer. Co., Hangzhou, China). *Alibaba Cluster Trace Program*. (2019). Accessed: Jan. 16, 2021. [Online]. Available: https://github.com/alibaba/clusterdata

[75] K. R. Rao, D. N. Kim, and J. J. Hwang, *Fast Fourier transform: Algorithms and Applications*. Dordrecht, The Netherlands: Springer, 2010, vol. 32.

[76] (Weaveworks Ltd., London, U.K.). *Container Solutions: Socks Shop—A Microservices Demo Application*. (2016). Online]. Available: https://microservices-demo.github.io/

[77] IBM Instana Team. "Sample microservices pplication running on Kubernetes." 2018. Accessed: Jan. 2, 2024. [Online]. Available: https://www.ibm.com/blog/sample-microservices-application-running-on-kubernetes/

[78] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "esDNN: Deep neural network based multivariate workload prediction in cloud computing environments," *ACM Trans. Internet Technol. (TOIT)*, vol. 22, no. 3, pp. 1–24, 2022.

[79] K. Rzadca et al., "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[80] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 3987–3995.

[81] A. A. Rusu et al., "Progressive neural networks," 2016, *arXiv:1606.04671*.

[82] P. Ruvolo and E. Eaton, "ELLA: An efficient lifelong learning algorithm," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 507–515.

[83] J. Yoon, E. Yang, J. Lee, and S. J. Hwang, "Lifelong learning with dynamically expandable networks," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–11.

[84] R. R. Karn, M. Ziegler, J. Jung, and I. A. M. Elfadel, "Hyper-parameter tuning for progressive learning and its application to network Cyber security," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2022, pp. 1220–1224.

[85] L. Zhu and N. Laptev, "Deep and confident prediction for time series at Uber," in *Proc. IEEE Int. Conf. Data Min. Workshops (ICDMW)*, 2017, pp. 103–110.

[86] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–35, 2022.

[87] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2114–2129, Sep. 2019.

[88] M. Xu et al., "Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster," *Softw., Pract. Exp.*, vol. 54, no. 1, pp. 39–57, Jan. 2024.

[89] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang,, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proc. 10th IEEE Int. Conf. Adv. Comput. Intell. (ICACI)*, 2018, pp. 583–588.

[90] S. Venticinque, R. Aversa, B. Di Martino, M. Rak, and D. Petcu, "A cloud agency for SLA negotiation and management," in *Proc. Eur. Conf. Parallel Process.*, 2010, pp. 587–594.

[91] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud-survey results and own solution," *J. Grid Comput.*, vol. 14, no. 2, pp. 265–282, 2016.
[92] F. Li, "Cloud-native database systems at Alibaba: Opportunities and challenges," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2263–2272, 2019.
[93] J. Rahman, "Building QoS-aware cloud services," Ph.D. Dissertation, Dept. Comput. Sci., Univ. Texas at San Antonio, San Antonio, TX, USA, 2019.
[94] Y. Hiroshima and N. Komoda, "Parameter optimization for hybrid auto-scaling mechanism," in *Proc. 17th IEEE Int. Symp. Comput. Intell. Inform. (CINTI)*, 2016, pp. 000111–000116.
[95] S. Luo et al., "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 412–426.
[96] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A microservice reference application for benchmarking, modeling and resource management research," in *Proc. 26th IEEE Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, 2018, pp. 223–236.
[97] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. 27th IEEE/ACM Int. Symp. Qual. Service (IWQoS)*, 2019, pp. 1–10.
[98] N. A. Simakov et al., "A workload analysis of NSF's innovative HPC resources using XDMoD," 2018, *arXiv:1801.04306*.
[99] R. R. Karn, P. Kudva, and I. A. M. Elfadel, "Dynamic autoselection and autotuning of machine learning models for cloud network analytics," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1052–1064, May 2019.
[100] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From security to assurance in the cloud: A survey," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1–50, 2015.
[101] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 674–691, Mar. 2021.
[102] J. Clark. "Raising data center power density." Oct. 2013. [Online]. Available: https://www.colovore.com/wp-content/uploads/2013/10/Colovore-Data-Center-Journal-10242013.pdf
[103] Z. Dong, W. Zhuang, and R. Rojas-Cessa, "Energy-aware scheduling schemes for cloud data centers on Google trace data," in *Proc. IEEE Online Conf. Green Commun. (OnlineGreenComm)*, 2014, pp. 1–6.
[104] M. Xu, A. N. Toosi, and R. Buyya, "A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing," *IEEE Trans. Sustain. Comput.*, vol. 6, no. 4, pp. 544–558, Oct.–Dec. 2020.
[105] P. Ruiu, A. Scionti, J. Nider, and M. Rapoport, "Workload management for power efficiency in heterogeneous data centers," in *Proc. 10th IEEE Int. Conf. Complex, Intell., Softw. Intensive Syst. (CISIS)*, 2016, pp. 23–30.
[106] N. Gholipour, E. Arianyan, and R. Buyya, "A novel energy-aware resource management technique using joint VM and container consolidation approach for green computing in cloud data centers," *Simul. Model. Pract. Theory*, vol. 104, Nov. s2020, Art. no. 102127.
[107] (Amazon Web Services, Seattle, WA, USA). *Amazon Elastic Container Service*. (2021). Accessed: Mar. 22, 2021. [Online]. Available: https://aws.amazon.com/ecs/
[108] (IBM, Armonk, New York, USA). *An Overview of IBM Bluemix*. (2021). Accessed: Mar. 22, 2021. [Online]. Available: https://www.ibm.com/support/pages/overview-ibm-bluemix
[109] (Google, Mountain View, CA, USA). *Containers on Compute Engine*. (2021). Accessed: Mar. 3, 2021. [Online]. Available: https://cloud.google.com/compute/docs/containers
[110] J. J. Dongarra, "Performance of various computers using standard linear equations software," *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 1, p. 17, 1990.
[111] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *Proc. IEEE Int. Symp. Workload Character. (IISWC)*, 2016, pp. 1–10.
[112] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proc. 5th Annu. Workshop Model., Benchmark. Simul.*, vol. 2011, 2009, p. 37.
[113] "The grid workloads datasets." Grid Workloads Archive. Accessed: Mar. 22, 2021. [Online]. Available: https://github.com/acmeair/acmeair
[114] W. Chen, K. Ye, Y. Wang, G. Xu, and C.-Z. Xu, "How does the workload look like in production cloud? Analysis and clustering of workloads on Alibaba cluster trace," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2018, pp. 102–109.
[115] "ACM air sample and benchmark." 2015. Accessed: Mar. 22, 2021. [Online]. Available: https://github.com/acmeair/acmeair
[116] A. Sekhar, B. Manoj, and C. S. R. Murthy, "A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks," in *Proc. Int. Workshop Distrib. Comput.*, 2005, pp. 63–74.
[117] W. Zheng and R. Sakellariou, "Budget-deadline constrained workflow planning for admission control," *J. Grid Comput.*, vol. 11, no. 4, pp. 633–651, 2013.
[118] Y. Hirashima, K. Yamasaki, and M. Nagura, "Proactive-reactive auto-scaling mechanism for unpredictable load change," in *Proc. 5th IIAI Int. Congr. Adv. Appl. Inform. (IIAI-AAI)*, 2016, pp. 861–866.
[119] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*. Sebastopol, CA, USA: O'Reilly Media, 2022.
[120] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2019, pp. 68–75.

Lamees M. Al Qassem received the M.Sc. and Ph.D. degrees from Khalifa University, Abu Dhabi, UAE, in 2017 and 2022, respectively, where she is currently a Postdoctoral Fellow. Her M.Sc. thesis focused on Arabic natural language processing (NLP) and artificial intelligence. Her Ph.D. thesis focused on microservice architecture and efficiency models for cloud computing services. She has published in the areas of educational technology, including augmented and virtual reality, Arabic natural language processing, cloud computing, blockchain, and microservice architectures. She received the Best Paper Award from the UAE 2019 Graduate Student Research Competition for her work on Arabic NLP.

Thanos Stouraitis (Life Fellow, IEEE) received the Ph.D. degree from the University of Florida. He is currently a Professor with the Department of Computer and Communications Engineering, Khalifa University, UAE, and a Professor Emeritus with the University of Patras. He has served on the faculties of The Ohio State University, the University of Florida, New York University, and The University of British Columbia. He also served on the National Scientific Board for Mathematics and Informatics of Greece. He was a Founding Council Member of the University of Central Greece. Along with several textbooks, he has authored about 200 technical papers, several book chapters, and holds one USA patent on DSP processor design. He has led several DSP processor design projects funded by the European Commission, American organizations, and the Greek government and industry. His current research interests include AI hardware systems, design and architecture of optimal digital systems with emphasis on cryptoprocessors, signal and image processing systems, and computer arithmetic. He received the IEEE Circuits and Systems Society Guillemin-Cauer Award. He has served as an Editor or a Guest Editor for numerous technical journals, including IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS. Also, as the General Chair and/or a Technical Program Committee Chair for many international conferences, including IEEE ISCAS, AICAS, SiPS, and ICECS. He has served IEEE in many ways, including as the President of its Circuits and Systems Society from 2012 to 2013. He is a Life Fellow of IEEE for his contributions to digital signal processing architectures and computer arithmetic.

Ernesto Damiani (Senior Member, IEEE) received the Honorary Doctorate degree from the Institut National des Sciences Appliquées, Lyon, France. He is the Director of the Center for Cyber-Physical System, Khalifa University, Abu Dhabi, and a Full Professor with the Department of Computer Science, Università degli Studi di Milano, Italy, where he leads SESAR Lab. He has held visiting positions at a number of international institutions, including George Mason University, Virginia, USA; Tokyo Denki University, Japan; and LaTrobe University, Melbourne, Australia. He has been a principal investigator in a number of large-scale research projects funded by the European Commission, the Italian Ministry of Research and by private companies, such as British Telecom, Cisco Systems, SAP, TIM, and Siemens Networks (currently Nokia Siemens). He has coauthored more than 700 scientific papers and many books, including *Open Source Systems Security Certification* (Springer 2009). His research interests include artificial intelligence, machine learning, cyber-physical systems, secure service-oriented computing, privacy-preserving big data analytics, and cyber-physical systems security. He is the Editor-in-Chief of the IEEE TRANSACTIONS ON SERVICE-ORIENTED COMPUTING and served as an Associate Editor for the IEEE TRANSACTIONS ON FUZZY SYSTEMS. In 2008, he was nominated ACM Distinguished Scientist and received the Chester Sall Award from the IEEE Industrial Electronics Society.

Ibrahim (Abe) M. Elfadel (Life Senior Member, IEEE) received the Ph.D. degree from the Massachusetts Institute of Technology in 1993. He is currently Professor of Computer and Communication Engineering with Khalifa University, Abu Dhabi, UAE. Before his current academic position, he was with the corporate CAD organizations at IBM Research and the IBM Systems and Technology Group, Yorktown Heights, NY, USA, where he was involved in the research, development, and deployment of CAD tools and methodologies for IBM's high-end microprocessors. From 2012 to 2019, he led three Abu Dhabi-based, industrially funded research centers dedicated to IoT, 3-D Integration, and MEMS. He is the recipient of six Invention Achievement Awards, one Outstanding Technical Achievement Award, and one Research Division Award, all from IBM, for his contributions in the area of VLSI CAD. His other awards include the D. O. Pederson Best Paper Award from the IEEE Transactions on Computer-Aided Design, the SRC Board of Directors Special Award for pioneering semiconductor research in Abu Dhabi, the Best Paper Award from the IEEE Conference on Cognitive Computing, Milan, Italy, in July 2019, and the 2022 Service Award from the International Federation of Information Processing. He is an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS. He has served on the Technical Program Committees of several leading conferences, including DAC, ICCAD, ASPDAC, DATE, ISCAS, AICAS, BioCAS, VLSI-SoC, ICCD, ICECS, and MWSCAS. He was the General Co-Chair of VLSI-SoC 2017, Abu Dhabi, UAE; the Technical Program Co-Chair of VLSI-SoC 2023, Sharjah, UAE; and AICAS 2023, Hangzhou, China. He will be the Technical Program Chair of CloudCom2024, Abu Dhabi.