Proactive and Reactive Auto-scaling Techniques for Edge Computing

32

Suhrid Gupta, Muhammed Tawfiqul Islam, and Rajkumar Buyya

Contents

32.1 Introduction . . . 595

32.2 Background and Key Concepts . . . 597
32.2.1 Service Level Agreements . . . 597
32.2.2 Micro-service Architecture . . . 597
32.2.3 Kubernetes Architecture . . . 598
32.2.4 Auto-scaling Overview . . . 599

32.3 Edge Computing Implementation . . . 599
32.3.1 Hierarchical Model . . . 599
32.3.2 Software-Defined Model . . . 599

32.4 Edge Computing Issues and Challenges . . . 600
32.4.1 Resource Allocation . . . 600
32.4.2 Cold Start Problem . . . 600
32.4.3 SLA Guarantees . . . 601

32.5 Reactive Auto-scaling Strategies . . . 601

32.6 Proactive Auto-scaling Strategies . . . 602

32.7 Hybrid Auto-scaling Strategies . . . 603

32.8 Experimental Case Study . . . 603
32.8.1 Experimental Setup . . . 603
32.8.2 Results . . . 604

32.9 Current Limitations . . . 604
32.9.1 Resource-Heavy Deployment . . . 604
32.9.2 Custom Metrics Support . . . 604
32.9.3 Lack of Initial Training Data . . . 604
32.9.4 Complex Hyper-parameter Tuning . . . 604

32.10 Future Work . . . 604
32.10.1 Multivariate Auto-scaling . . . 604
32.10.2 Multi-SLA Constraints . . . 605

32.11 Summary and Conclusions . . . 606

References . . . 606

S. Gupta · M. T. Islam (✉) · R. Buyya
Cloud Computing and Distributed Systems (CLOUDS) Lab School of Computing and Information Systems, The University of Melbourne, Melbourne, VIC, Australia
e-mail: suhrid.gupta1@unimelb.edu.au; tawfiqul.islam@unimelb.edu.au; rbuyya@unimelb.edu.au

Abstract

Edge computing allows for the decentralization of computing resources. This decentralization is achieved through implementing micro-service architectures, which require low latencies to meet stringent service level agreements (SLAs), such as performance, reliability, and availability metrics. While cloud computing offers the large data storage and computation resources necessary to handle peak demands, a hybrid cloud and edge environment is required to ensure SLA compliance. Several auto-scaling algorithms have been proposed to try and achieve these compliance challenges, but they suffer from performance issues and configuration complexity. This chapter provides a brief overview of edge computing architecture, its uses, benefits, and challenges for resource scaling. We then introduce service level agreements and existing research on devising algorithms used in edge computing environments to meet these agreements, along with their benefits and drawbacks.

Keywords

Cloud computing · Edge computing · Service level agreements · Auto-scaling

32.1 Introduction

Cloud computing architectures leverage the on-demand accessibility of the Internet. The applications deployed here utilize the vast resources of the cloud to perform a task and relinquish it once it is complete for the other sub-modules in the application to request [1]. In the early days, a singular end point would be used to access these services; however, nowadays the architecture is multiregional, allowing effortless access from across the world. This was achieved through the

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026

F. Xhafa (ed.), *Springer Handbook of Data Engineering*, Springer Handbooks,
https://doi.org/10.1007/978-3-031-92372-2_32

use of content delivery networks (CDNs) located in several regions to allow for data to be quickly replicated and served to clients. This architecture model allows for the processing of large-scale data in a near real-time manner.

During the early twenty-first century, this architecture paradigm dominated the information technology (IT) industry. Compared to traditional monolithic architectures, the ease of deployment, and scalability, coupled with the economic benefits ensured its dominance. The increasing popularity of handheld devices as well as home appliances has resulted in data being largely produced at the edge of the cloud network. Thus, processing this large amount of data solely on the cloud proved to be an inefficient solution due to the bandwidth limitations of the network [2]. To counteract this inefficiency, edge computing paradigms were built on the previous foundation of CDNs [3]. Edge computing architectures ensure data processing services and resources exist at the peripheries of the network [4]. The architecture extends and adapts the computing and networking capabilities of the cloud to meet real-time, low-latency, and high-bandwidth requirements of modern agile businesses.

Edge computing deploys several lightweight computing devices known as *cloudlets* to form a "mini-cloud" and places them in close proximity to the end-user data [5]. This reduces the latency in terms of client-server communication and data processing. Figure 32.1 shows a high-level overview of this architecture. Cloudlets can also be easily scaled depending on the resource requirements per edge architecture [6]. However, due to the dynamic resource requirements that may fluctuate from time to time, the resources allocated to cloudlets must be dynamically scaled too. This dynamic scaling, along with the inherent latency present between the cloud layer and the edge cloudlets, poses a significant problem to real-time resource scaling [7].

One method of mitigating this scaling latency is through the use of micro-service applications. By employing a micro-service architecture, the resources in a cloudlet are distributed as a collection of smaller deployments that are both independent and loosely coupled [8]. This loose coupling ensures that parts of the cloudlet can be scaled as required, further reducing the time required to scale resources as compared to scaling the cloudlet monolithically.

The scaling of these micro-service resources is done automatically through a process known as auto-scaling. While most container orchestration platforms come bundled with default auto-scaling solutions, and these solutions are sufficient for most applications, they fall apart when scaling resources for time-sensitive services processing real-time data such as the ones used in healthcare require stringent compliance to service level agreements (SLAs) on metrics such as application latency. This has led to further research on auto-scaling solutions for edge computing applications. These primarily fall into two categories. Reactive auto-scaling solutions attempt to modify the micro-service resource allocation

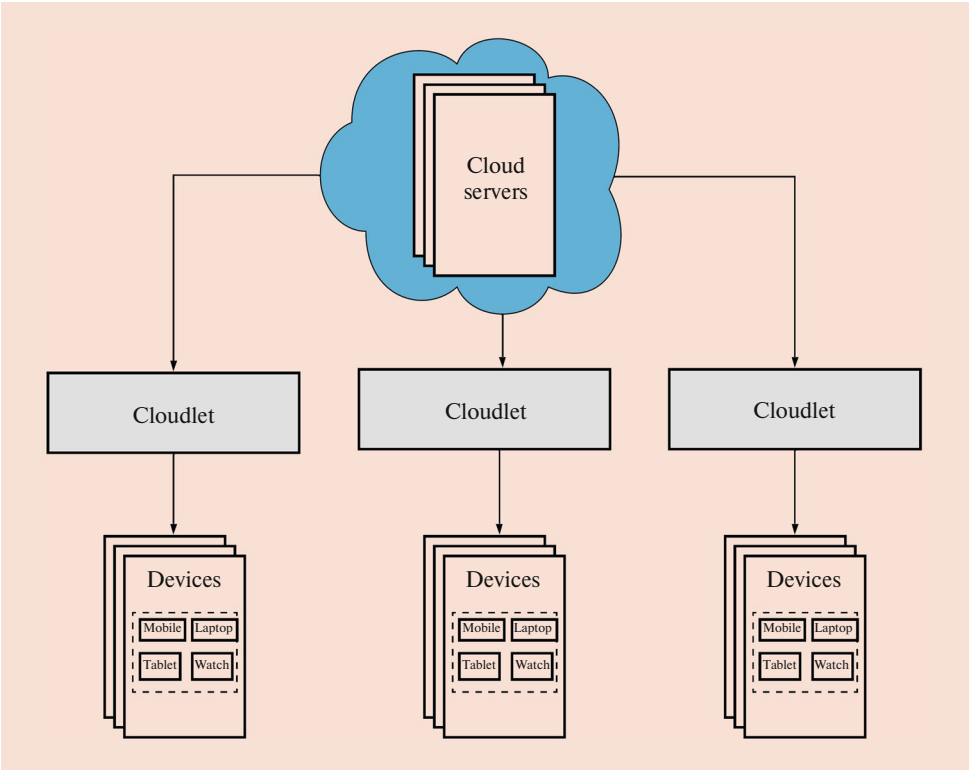

Fig. 32.1 Overview of edge computing architecture

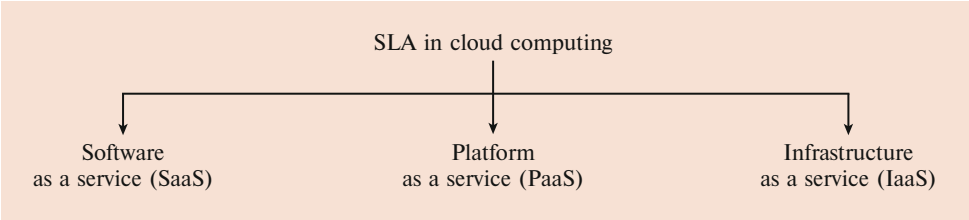

Fig. 32.2 Overview of service level agreements

once the required resources exceed the current allocation. These algorithms are simple to develop and deploy; however, the time taken to scale resources leads to a degradation of resource availability and violates SLA compliance [9]. To counteract these pitfalls, proactive auto-scaling solutions attempt to model resource allocation over time and effectively predict the resource requirements. By doing so, the micro-service resources can be scaled in advance through a process known as "cold starting." This approach removes the latency inherent in scaling resources; however, the algorithms are extremely complex to develop, train, and tune to specific edge applications [10].

32.2 Background and Key Concepts

In this section, a brief introduction to service level agreements is provided, followed by an overview of micro-service architectures. Finally, we give a brief description of the architecture of Kubernetes and its auto-scaling algorithms.

32.2.1 Service Level Agreements

Cloud computing generally exposes resource using a pay-as-you-go service. These lucrative plans have led to the implementation of applications and hardware being delivered as software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS), as shown in Fig. 32.2. However, consumers of such services have demands that may vary significantly, and it is impossible to fulfill all these expectations. Thus, a balance needed to be struck in order to commit to an agreement [11].

This commitment is known as a service level agreement (SLA). This SLA defines the expected services provided by the provider and agreed to by the consumer. For example, one of the most common metrics by which SLAs are negotiated between providers and consumers is the availability of service.

Availability of Services

Availability is defined to ensure that the functional performance of the edge deployment is maintained for an agreed period. SLAs mostly define either monthly or yearly downtime in order to compute service credits for billing purposes [12]. The downtime can be calculated using the following formulae:

$$downtime_{monthly} = \frac{100 - Availability\%}{100} \times 30 \times 24 \quad (32.1)$$

$$downtime_{yearly} = \frac{100 - Availability\%}{100} \times 365 \quad (32.2)$$

Table 32.1 Summary of SLA availability

Availability %	Monthly downtime	Yearly downtime
90%	72 h	36.5 days
99%	7.2 h	3.65 days
99.9%	43.8 min	8.76 h
99.99%	4.38 min	52.56 min
99.999%	25.9 s	5.26 min

Table 32.1 shows the expected downtimes for several SLA availability percentages.

32.2.2 Micro-service Architecture

Micro-service architectures involve decomposing an application into several loosely coupled services and deploying them on separate cloudlet servers known as "nodes." These services communicate with each other through a lightweight framework such as RESTful APIs [13]. Within these services, application data and commands are stored and executed within "containers." Typically, these architectures provide scalability, as well as ease of deployment and modification. Availability, however, remains an important concern for such deployments. For a deployment to be classified as "highly available," it must be accessible at least 99.999% of the time. For example, a highly available search engine would only face 5 min of downtime per year [14]. Therefore, an orchestration mechanism is required to manage the deployment and communication of these containers.

Container orchestration allows the micro-service application to customize how deployment, monitoring, and

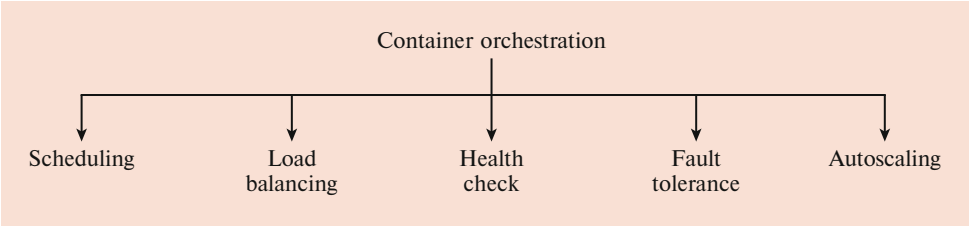

Fig. 32.3 Features of container orchestration

controlling function [15]. Figure 32.3 depicts the typical features of container orchestration.

Scheduling defines the rules on the number of containers to be executed at any given time. Scheduling also places containers on specific nodes based on availability and best performance.

Load balancing distributes the resource usage among multiple micro-service nodes. By default, a round-robin policy is implemented, although more complex policies may be implemented at the discretion of the developer.

Health checks ensure that the container is still capable of responding to queries. Typically, these are done using a periodic lightweight HTTP request and verifying the response.

Fault tolerance maintains several replicas of containers, a strategy commonly used to achieve the high availability mentioned above. Health checks are used to ensure the replicas are functioning, and they typically have strategies to ensure there is no mismatch in data between two fault tolerant containers.

Auto-scaling is the process of automatically adding or removing resources or containers. Internal metrics such as CPU usage are typically used; however, custom policies can also be implemented at the discretion of the developer.

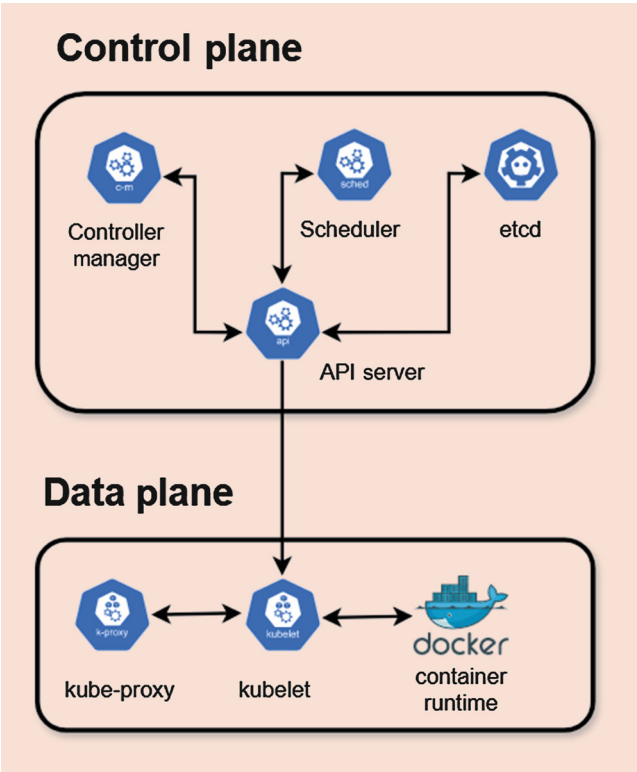

Fig. 32.4 Overview of Kubernetes architecture

32.2.3 Kubernetes Architecture

Kubernetes is one of the most popular open-source container orchestration platforms [16]. Initially referred to as "Borg," the project was used internally at Google to deploy the majority of their cloud applications before becoming an open-source application [17]. Figure 32.4 shows the high-level architecture. The Kubernetes deployment has a controller/worker architecture. The nodes in the Kubernetes cluster are split into either *control plane nodes* or *data plane nodes*. The *control plane nodes* have a collection of processes that help monitor and maintain the desired state of the deployment. The *data plane nodes* contain processes that run the containers doing the actual work and are managed by the control plane.

The smallest unit of work in a Kubernetes deployment is known as a *pod* [18]. This is a collection of containers sharing an IP address and port. In summary, micro-service architectures are said to be containerized and deployed on Kubernetes in the form of pods [16].

Control Plane

The *API server* is the primary communication endpoint for the entire deployment. Every component in the architecture communicates through it to exchange information. It is also used to update the current deployment state. The API server is a simple RESTful API implementation, exposing well-documented APIs for access by other components as well as developers. Multiple replicas of this component are typically maintained to ensure high availability.

The *etcd* is a data store that persists the deployment state in a key-value format. The data is serialized unlike in the stateless API server. This data adheres the properties of *recovery* and *availability*. *Recovery* ensures that any corruption of data

is reverted using a system of backups, such as checkpoints. *Availability* ensures that the deployment is reachable by the end user regardless of the traffic being requested on the network.

Controller manager implements the desired deployment state. During initial deployment, the controller manager inputs the required workload as the desired state, after which it continually monitors the deployment state using a system of looping controls. If the deployment requires modifications, they are achieved using the API server, and the deployment is brought back into alignment with the desired state.

Finally, the *scheduler* decides the location where the pod will be deployed. The scheduler runs a control loop that searches for unscheduled pods using the API server. It then assigns the pods to a data plane node based on several predicates and priorities, such as resource requirements and node affinity, respectively.

Data Plane

The *container runtime* is a process that downloads or "pulls" the image for the required container onto the node. Kubernetes supports a wide range of runtimes, but some of the popular solutions are CRI-O, containerd, and Docker.

The most important process running on every data plane node is the *kubelet*. This process executes the image assigned to the node via the container runtime, performs health checks, and reports the node status to the control plane.

Another data plane process is the *kube-proxy*, which manages the rules for forwarding requests to services, as well as the IP tables of nodes. If a service is added or removed, kube-poxy updates the IP table accordingly.

32.2.4 Auto-scaling Overview

The process of scaling nodes, pods, or other resources depending on requirements in an automated manner is known as *auto-scaling*. Typically, there are three variations of auto-scaling.

Cluster auto-scaling modifies the number of nodes running in the entire deployment or cluster. Dynamically allocating nodes based on resource requirements helps to manage the cost of running the deployment. The autoscaler works by looping through two tasks. The first watches for unscheduled pods, while the second checks if the current deployed pods can be merged on a smaller number of nodes.

Vertical pod auto-scaling modifies the CPU and memory resources assigned to pods. By default, the scheduler reserves a larger amount of these resources to pods than is usually required. By performing vertical pod auto-scaling, the cluster can better manage its over-provisioned resources in real time.

Horizontal pod auto-scaling is the most commonly used auto-scaling strategy [19]. It modifies the number of pods assigned to a task, based on the resources being requested. A periodic control loop compares the actual resource utilization with the target utilization defined by the initial deployment setup and scales the number of pods accordingly.

32.3 Edge Computing Implementation

Yu et al. [20] states that when implementing edge computing architectures, researchers typically focus on two models, namely, the hierarchical model and the software-defined model.

32.3.1 Hierarchical Model

Since edge and cloud servers are deployed at varied distances from the end users, it makes sense that the edge architecture is to be divided into such a hierarchy, with each layer defining the functions relative to resource availability and distance.

Smeliansky [21] gives an overview of such a hierarchical model. The bottom tier is usually comprised of a geo-distributed servers that receive workload from mobile devices via wireless links. These servers are then connected to a higher tier level of servers that comprise remote data centers. If the mobile workload received by a lower tier server exceeds its computational capacity, it can offload this excess workload to a higher tier server. In this way, the architecture can serve large load peaks.

Several research efforts exist for such a model. Tong et al. [22] proposed an edge cloud hierarchy that could be used to serve high load demands from mobile users. In such a model, the cloudlet servers were deployed at the edge layer, while the cloud was established as a tree hierarchy. By using such an implementation, the edge layer was able to aggregate its servers' resource abilities to better serve peak workloads.

Jarwah et al. [23] demonstrated a similar approach to a hierarchical model, which integrated mobile edge computer (MEC) servers and cloudlets. The mobile users obtained specific services as per requested, and the MEC served the ability to deliver their computational and storage requirements.

32.3.2 Software-Defined Model

The number of end users and devices that connect to the edge architecture typically numbers in the millions [20]. Thus, management of an application deployed on this architecture can prove to be significantly challenging. To address these complexities, software-defined networks (SDNs) were proposed.

According to Wang et al. [24], SDNs distinguish themselves from conventional networking approaches by

decoupling the control plane from the data plane. The control plane is constructed using a combination of dedicated controllers. These serve as the control center of the SDN model, while the data plane simply forwards data packets in the form of a network switch. Such a decoupling makes the architecture highly flexible, as well as simplifies network management [25]. However, they have drawbacks, which include performance issues if not developed properly.

Despite these challenges, there have been a number of research efforts based on the SDN model. Du and Nakao [26] presented an MEC model that was application specific. In this model, the software-defined data plane acts as a mobile virtual network operator (MVNO). The mechanism computes a hop-count-based tethering as well as optimizes the process. Fairness among determining user resources is determined through regulating concurrent TCP connections.

Similarly, Jaraweh et al. [27] proposed an SDN model that integrated its capabilities to the MEC system. The management and administrative requirements for the entire model could be reduced in this manner.

Finally, Salman et al. [28] demonstrated an integration of SDN, MEC, and a network function virtualization (NFV)m, which was capable of achieving better MEC performance in mobile networks than the other two works. Such a model could be further extended to enable an IOT-capable deployment.

32.4 Edge Computing Issues and Challenges

Maintaining SLA compliance in an edge computing architecture poses several unique challenges. In this section, we briefly describe a few main issues.

32.4.1 Resource Allocation

Cao et al. [4] demonstrated the key differentiations traditional cloud computing architectures have compared to edge architectures while asserting that edge deployments remain an extension of the cloud. The aim of cloud computing infrastructures is to process huge amounts of data from multiregional zones or, in the best case, globally. This is done so as to perform in-depth analysis in diverse fields, such as healthcare, robotics, and business decision-making. Traditionally, they also dealt with non-real-time data for decision-making [29]. On the other hand, edge computing usually handles smaller-scale data that is locally clustered and isolated in separate zones and highly real time in nature [30]. The data processed in traditional cloud computing environments are also generally done using a high network bandwidth. This is due to the large distances data needs to be transmitted over to reach the data centers and cloud servers. Such data transmission places an enormous burden on the cloud network and poses multiple security challenges in ensuring that the data is not compromised in transit.

The real-time nature of edge computing applications necessitates a method of resource allocation, which ensures minimal cost of deployment and maximum efficiency in terms of performance. Micro-service container orchestration technologies are leveraged to achieve these aims. Kristiani et al. [31] demonstrated an edge computing architecture, where the edge layer consists of Kubernetes nodes. Such a deployment increases the scalability, as well as maintains the ease of deployment, upgrade, and removal of nodes in the edge layer. Scaling of resources through the means of auto-scaling depending on the resource requirements is crucial to the architecture's performance. Default solutions, such as the inbuilt autoscaler provided by Kubernetes, while generally useful for cloud applications, are unsuitable for edge architectures according to Phan et al. [32]. They note that due to the algorithm's default nature to allocate resources in a round-robin manner, they do not take into account which Kubernetes nodes require priority resource allocation, violating edge architecture paradigms.

32.4.2 Cold Start Problem

To explain the cold start problem, we use the example of horizontal pod auto-scaling in Kubernetes. When the control plane requests for a deployment replica to be scaled up, Kubernetes adds more pods to the data plane nodes. Based on the internal workload, the pod needs to be elastically scaled out [33]. Even though the pod's startup time is significantly quicker than, say, a traditional virtual machine, there is a latency inherent to bootstrapping the container, preparing the pod environment based on the deployment specification, and initializing the code present in the container image, and registering the pod in a "ready" state to the Kubernetes control plane.

Several techniques exist to mitigate this resource latency. The Kubernetes container runtime uses snapshots [34], lazy fetching of container images [35], and container queues [36]. However, these measures do not eliminate the issue of the inherent latency in installing and registering the resource. Due to this, researchers looked into scaling resources in a predictive manner, so as to ensure the micro-service application has enough time to spool up resources before the actual demand comes in. This process, by which resources are created and registered with the container orchestration before the expected workload, is known as resource *cold start*.

32.4.3 SLA Guarantees

There are several challenges posed in providing SLA guarantees in an edge deployment:

- Users queueing for large periods of time to use a service [37]
- Degradation of application performance due to peak levels of workload, leading to user dissatisfaction [38]
- Incorrect resources being allocated to the application, leading to either a degradation of availability, or large cost of application deployment [39]

Several strategies have been proposed to counteract these challenges. Linlin et al. [40] proposed a customer-driven strategy to minimize the provisioning costs. The algorithm considers the customer profiles as well as cloud providers' quality parameters, such as response time to dynamically handle customer requests. Rajkumar et al. [41] proposed a solution for alleviating the issue of delay in service allocation to users through the use of a novel hierarchical scheduling algorithm. This algorithm increases the performance of the scheduling algorithm, thus reducing the wastage of resources and minimizing wait times. Sakr et al. [38] introduced a novel approach to combat application performance degradation by using a middleware between consumers and the cloud. This middleware helps to facilitate dynamic provisioning of cloud databases based on consumer requirements, tailoring their needs and requirements to mitigate peak usages being hit often.

32.5 Reactive Auto-scaling Strategies

Nunes et al. [42] stated that horizontal pod auto-scaling using a reactive strategy remains the most popular auto-scaling technique, as well as research topic. These strategies, despite having limitations, such as reliance on predetermined resource thresholds and delays in resource scaling, have been popular in research articles. Dogani et al. [43] stated that this was due to the simplicity and user-friendliness in developing them.

Kampars and Pinka [44] proposed a reactive auto-scaling algorithm for edge architectures based on open-source technologies. The algorithm scales in a nonstandard approach, considering real-time adjustments in the application logic to determine the strategy of scaling, resulting in several improvements in performance.

Zhang et al. [45] presented an algorithm for determining edge elasticity through container-based auto-scaling. The authors posit that elasticity is a key factor of how an edge deployment, as well as the lightweight containers that make up the edge layer perform. The framework not only autoscales container resources but also monitors resource usage. They were able to show experimentally that balancing system stability with a decent elasticity required careful tuning of parameters, such as the cooldown periods of scaling.

Srirama et al. [46] investigated a container-aware auto-scaling solution that deploys applications to containers it deems "best fit." The algorithm also uses a rule-based policy to minimize the deployment time, thus mitigating the issue of cold start. Finally, a dynamic bin-packing sub-algorithm ensures that the applications are deployed on the least required physical servers, thus minimizing wastage of computing resources. The authors experimentally demonstrated that this algorithm minimized the processing time, cost, and resource utilization.

Hoenisch et al. [47] implemented a fourfold auto-scaling strategy for containerized applications, which asks if the containers or servers can be autoscaled horizontally or vertically. This question is formalized as a multi-objective optimization problem, and the approach used reduced the cost of each request by more than 20%.

Santos et al. [48] implemented a quality of experience based auto-scaling of containerized edge deployments. The algorithm can autoscale both horizontally and vertically on a set of quality metrics that can be customized by the end user. The authors explained that the experimental results displayed a performance comparable to other reactive solutions.

Sheganaku et al. [49] devised a container-based auto-scaling solution that allocates resources in a fourfold manner similar to Hoenisch et al. [47]. The authors formulated the problem as a multi-objective optimization problem and applied a mixed-integer linear programming (MILP) approach to allocate resources to containers. Such an approach demonstratively reduced costs while maintaining SLA constraints.

Taherizadeh and Stankovski [50] proposed a multi-level auto-scaling solution using a rule-based approach. The algorithm uses dynamically changing thresholds based on both the container infrastructure and application, resulting in improved performance as compared to other reactive approaches.

Phan et al. [32] proposed a reactive auto-scaling solution for edge deployments of IoT devices that dynamically allocates resources based on incoming traffic. This traffic-aware horizontal pod autoscaler (THPA) operates on top of the underlying Kubernetes architecture. As discussed above, the default Kubernetes horizontal pod autoscaler scales resources in a round-robin manner, not taking into context which nodes are receiving the highest resource requests. THPA alleviates this issue by modeling the resource requests per Kubernetes nodes. It then intelligently allocates pods to the nodes with higher number of requests. The authors were able to experimentally demonstrate that following such an approach provided a 150% improvement in response time and throughput. Table 32.2 provides a comparative summary of reactive auto-scaling solutions.

Table 32.2 Summary of reactive auto-scaling solutions

	Reactive algorithms							
Features	[32]	[44]	[45]	[46]	[47]	[48]	[49]	[50]
Simple parameter tuning	✓	✗	✓	✗	✓	✓	✓	✗
Custom metrics	✗	✗	✗	✗	✗	✓	✓	✗
Lightweight deployment	✓	✓	✓	✓	✗	✗	✗	✓
Edge architecture compliant	✓	✗	✓	✓	✓	✓	✓	✓
SLA compliant	✗	✗	✗	✗	✗	✗	✗	✗

Table 32.3 Summary of proactive auto-scaling solutions

	Proactive algorithms					
Features	[54]	[55]	[56]	[57]	[59]	[60]
Simple parameter tuning	✗	✗	✗	✓	✗	✗
Custom metrics	✓	✓	✗	✗	✗	✗
Lightweight deployment	✗	✗	✗	✓	✗	✗
Edge architecture compliant	✓	✗	✗	✗	✗	✗
SLA compliant	✗	✓	✓	✓	✗	✗

32.6 Proactive Auto-scaling Strategies

Lorido et al. [51] showed that compared to reactive algorithms, proactive algorithms achieved better resource allocation once they had been carefully optimized. Machine learning (ML) techniques, such as auto-regressive integrated moving averages (ARIMA) and long short-term memory (LSTM), have gained popularity in time-series analysis due to their relative ease of building and efficiency compared to other ML models. Through the careful use of these models, linear patterns in the data can be automatically identified in a short amount of time with relatively constrained resources. There are, however, several challenges when implementing a proactive algorithm. Time-series analysis models may struggle when dealing with highly complex and nonlinear data [43]. The development of a generalized algorithm for several edge architectures remains a costly process. One of the biggest challenges is the initial lack of training data. Another issue is the exploding or vanishing gradient problem [52], though modern algorithms ensure that they avoid this pitfall [53]. Despite these challenges, their application in scaling of resources with semi-predictable data series remains valuable.

Ju et al. [54] presented a proactive horizontal pod autoscaling solution for edge computing paradigms. The algorithm, known as proactive pod autoscaler (PPA), was designed to predict resource requests on multiple user-defined metrics, such as CPU request and I/O traffic requests. The algorithm does not use any specific machine learning model for the time-series analysis; instead, the model is to be inputted by the user. This model agnostic architecture allows for a very high level of customization. The user can deploy an ARIMA, LSTM, or even Bayesian confidence models. In a confidence model, the autoscaler will only deploy resources if the confidence value is seen to be above a specified user-defined threshold. The authors validated their findings by testing the architecture using LSTM and ARIMA models, and the results concluded that this algorithm significantly outperformed both the default Kubernetes autoscaler and the existing reactive auto-scaling solutions.

Meng et al. [55] created a proactive auto-scaling algorithm for forecasting the Kubernetes CPU usage of containers using a time-series prediction. They achieved this using the ARIMA model. The time-series was split into a training and validation set using a 5:1 ratio before being passed to the deep learning model. The authors were able to demonstrate experimentally that such an architecture reduced the forecast errors to 6.5%, as compared to the baseline of 17%.

Imdoukh et al. [56] proposed a proactive auto-scaling solution using an LSTM model, designed for edge computing architectures. The algorithm uses an LSTM neural network to predict future network traffic workload to determine the resources to assign to edge nodes ahead of time (cold start). The authors experimentally demonstrated that their algorithm was as accurate as existing ARIMA-based proactive solutions, but theirs significantly reduced the prediction time, as well as computed the minimum resource allocation required to handle future workload.

Messias et al. [57] created a proactive autoscaler using genetic algorithms (GAs). The genetic algorithm combines several time-series forecasting models while having the benefit of not requiring a training phase as the model adapts to the incoming data. The experimental results concluded that this approach produces results comparable to several state-of-the-art proactive models and can adapt to various time-series models.

Abdulla et al. [59] devised an auto-scaling solution that is capable of detecting sudden bursts in dynamic workloads. The algorithm achieves this through a method of workload and resource prediction to make a scaling decision. Experimenting on several burst-heavy workloads, the autoscaler demonstrated significant improvements compared to other state-of-the-art methods.

Alidoost et al. [60] proposed a workload classification model using a support vector machine (SVM). The algorithm extracts the user's workload characteristics and then trains the SVM on it. The authors demonstrated a 10% forecast error reduction compared to other machine learning proactive forecast approaches. Table 32.3 provides a comparative summary of proactive auto-scaling solutions.

32.7 Hybrid Auto-scaling Strategies

All the approaches mentioned in Sects. 32.5 and 32.6 have their benefits and drawbacks. Thus, hybrid solutions that merge multiple auto-scaling methods were proposed [61]. While hybrid algorithms for cloud-based deployments exist, integrating them into edge architectures poses several challenges due to the lower data storage and computational capacity of the edge layer. Furthermore, extracting the proactive time-series analysis to the cloud layer poses further challenges due to the inherent latency present between the two layers.

In 2007, one of the first hybrid algorithms for a distributed deployment was proposed by Jing et al. [62]. This algorithm combined rule-based fuzzy inference with machine learning forecasting for dynamic resource allocation. The authors experimentally verified their algorithm through a prototype to demonstrate that it can reduce the resource consumption on resource management systems compared to their default resource allocation algorithms.

Based on this work, Lama and Zhou [63] proposed a resource provisioning algorithm for multi-cluster setups using a hybrid autoscaler. The autoscaler comprised a combination of fixed fuzzy rule-based logic and a self-adaptive algorithm that dynamically tuned the scaling factor. The authors tested this algorithm on a simulation to demonstrate performance benefits compared to existing approaches.

A hybrid approach for cloud computing architectures was proposed by Rampérez et al. [64]. The algorithm, called forecasted load auto-scaling (FLAS), combines a predictive model for forecasting time-series resources, while the reactive model estimates other high-level metrics and delegates for the proactive model, reducing the potential forecast error when encountering previously unseen workloads. The approach was shown to demonstrate efficient resource allocation as compared to other state-of-the-art solutions.

Biswas et al. [65] presented a hybrid algorithm designed for cloud computing deployments with service level agreements. The proactive algorithm involves a machine learning-based approach using an SVM model, alongside the reactive algorithm to dynamically allocate resources. The algorithm was experimentally shown to perform better than a pure reactive or proactive solution in most cases.

Finally, another cloud computing-based autoscaler with SLA constraints was proposed by Singh et al. [66]. The robust hybrid autoscaler (RHAS) was designed particularly for web applications. The reactive rule-based autoscaler deals with current workload requirements, while the proactive model attempts to predict incoming resource workloads. The proactive forecaster uses a modification of the ARIMA machine learning model, known as the technocrat ARIMA and SVR model (TASM) [67]. The authors tested their algorithm on two data-sets belonging to ClarkNet and NASA. The technique was demonstrated to both reduce cloud deployment cost and SLA violations while giving consistent CPU utilization. Table 32.4 provides a comparative summary of hybrid auto-scaling solutions.

Table 32.4 Summary of hybrid auto-scaling solutions

	Hybrid algorithms				
Features	[62]	[63]	[64]	[65]	[66]
Simple parameter tuning	✓	✓	✓	✗	✗
Custom metrics	✓	✗	✗	✓	✓
Lightweight deployment	✓	✗	✓	✗	✗
Edge architecture compliant	✗	✗	✗	✗	✗
SLA compliant	✗	✗	✓	✓	✓

32.8 Experimental Case Study

We propose a novel hybrid auto-scaling algorithm for SLA-constrained edge computing applications, combining a machine learning-based proactive forecaster with a reactive autoscaler for real-time adjustments. Integrated into Kubernetes, the algorithm was extensively evaluated in edge environments, demonstrating superior SLA compliance and resource efficiency compared to existing solutions.

32.8.1 Experimental Setup

The experimental evaluation of the hybrid autoscaler was conducted using a micro-service architecture deployed on Kubernetes, leveraging its configurability and suitability for edge deployment scenarios. The DeathStarBench social network application was chosen as the testbed, simulating large-scale social network functionalities, such as user registration, timeline management, and content interaction. The architecture incorporated NGINX as the load balancer and MongoDB for persistent storage. The design was structured with distinct logic and storage layers, allowing for the realistic simulation of dynamic workloads typical in real-world edge deployments.

Workload Type and Generation The workload represented IoT application usage patterns with daily cycles, including peaks in the morning and evening. These patterns were emulated using the *wrk2* HTTP generator, which was customized with an algorithm introducing randomness to reflect real-world variations. The requests focused on the *home-timeline-service*, a critical component responsible for retrieving user timelines. Approximately 2.55 million GET requests were generated over 5 days, ensuring adequate data for a comprehensive evaluation of the autoscaler's performance.

Cluster Configuration The cluster consisted of six virtual machines deployed on the Melbourne Research Cloud, configured into cloud and edge layers. The cloud layer included two high-resource VMs for the Kubernetes control plane and database storage, while the edge layer comprised four low-resource VMs to simulate the resource constraints typical of edge deployments. Simulated latency was added between the cloud and edge layers to represent geographical separation. Kubernetes v1.28.2 was used with Flannel as the inter-pod communication layer, and Prometheus, integrated with Jaeger, provided system monitoring and latency tracking.

Baseline Algorithms Two baseline autoscalers were compared to the hybrid solution:

- **Default Kubernetes Horizontal Pod Autoscaler (HPA)**: Reactive scaling with no workload awareness, configured with a 50% CPU utilization threshold. It applied round-robin scheduling and suffered from significant cold start delays.
- **Hybrid Autoscaler**: The proposed solution combines reactive and proactive strategies, employing an LSTM-based forecaster trained on noise-reduced, pre-processed time-series data for anticipatory scaling, complemented by reactive scaling for immediate response.

32.8.2 Results

The *home-timeline-service* workload evaluation demonstrated the hybrid autoscaler's superiority in maintaining service quality under dynamic conditions, outperforming the default algorithm in request latency and SLA compliance.

Request Latency Latency analysis revealed the hybrid autoscaler's capability to eliminate cold start delays, a critical limitation of reactive strategies. The average latency remained under 30 ms, well within the flexible SLA threshold of 150 ms. In contrast, the default HPA exhibited frequent latency spikes exceeding 300 ms. Figure 32.5 compares the request latencies of the hybrid and default autoscalers, highlighting the hybrid solution's consistent performance.

32.9 Current Limitations

32.9.1 Resource-Heavy Deployment

Several auto-scaling solutions discussed here have been created for cloud computing architectures. Thus, deploying them to an edge architecture poses further challenges, as they may occupy large portions of the limited available resources. When this occurs, it becomes more difficult to maintain SLA compliance while keeping deployment costs down.

32.9.2 Custom Metrics Support

A lot of the algorithms are hard coded to use certain metrics to perform auto-scaling, for example, CPU and memory usage. While this makes the algorithm easier to deploy, it restricts its customization capabilities, since different deployments may wish to autoscale on different user metrics such as I/O load.

32.9.3 Lack of Initial Training Data

Proactive autoscalers rely on large amounts of training data to accurately predict the resource requirements for the deployment. This poses an issue when the autoscaler is initially deployed, as such data may not exist. During this period, the autoscaler is not guaranteed to output accurate results, damaging its SLA compliance and thus making it unsuitable for edge computing. Hybrid autoscalers attempt to remedy this issue by falling back to the reactive component while the initial data is being generated.

32.9.4 Complex Hyper-parameter Tuning

A problem exclusive to proactive and hybrid autoscalers is the hyper-parameters, which come bundled with the algorithms. These hyper-parameters require delicate tuning before they can be deployed in a real world setup. Failure to do so may result in erroneous results, jeopardizing the SLA compliance of the deployment.

32.10 Future Work

32.10.1 Multivariate Auto-scaling

The forecasters used in most proactive and hybrid autoscalers are a univariate model. This means that at any given time, only one variable is changing. Therefore, the input is a one-dimensional array. An extension can be made to the forecaster to convert it into a multivariate model. In this approach, multiple variables are being modified at a given time; thus, the input to the model is a more complex two-dimensional array.

The benefit of a multivariate approach is that a more holistic auto-scaling decision can be made using this. The drawback of such an approach is that the proactive forecaster training times will significantly increase due to the additional complexity of the input as well as output.

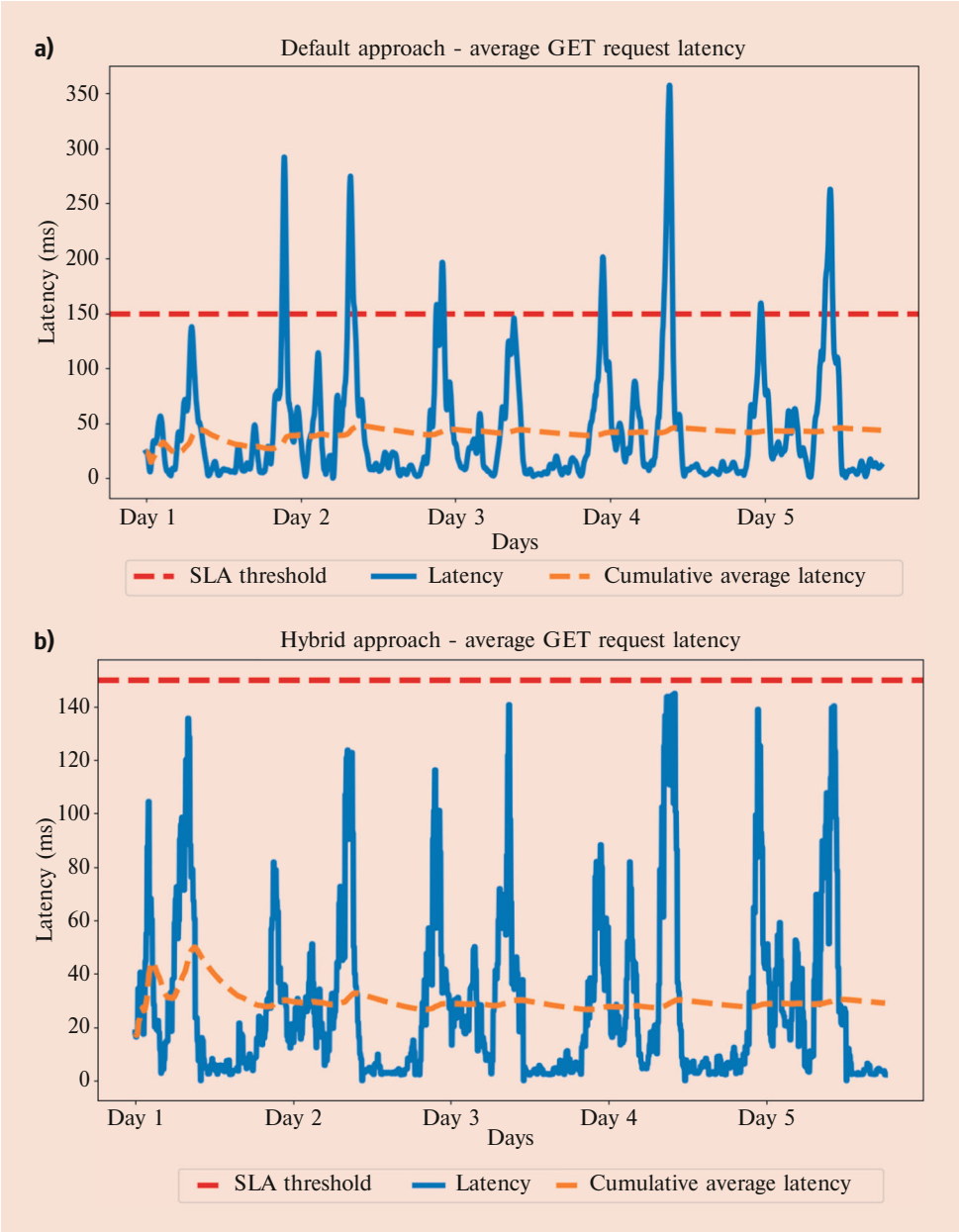

Fig. 32.5 Comparison of request latency between the default HPA and hybrid autoscaler under dynamic workloads. (**a**) Request latency using default HPA. (**b**) Request latency using hybrid autoscaler

32.10.2 Multi-SLA Constraints

Another enhancement that can be made is to support the use of multiple SLA constraints. By modifying the autoscaler into a multiparameter model, it can keep a track of several metrics and, if a violation occurs on any one of them, can tune the auto-scaling process accordingly.

Such a modification opens up several other possibilities too. Each SLA metric can be given its own weight to denote importance. The current hybrid autoscaler only considers a single action of tuning the forecaster hyper-parameter variables on SLA violation occurrences. By making it a multi-SLA model with different weights attached, the autoscaler can intelligently decide what actions to take when on different scenarios, such as urgent and mild SLA violations.

This adds further complexity to the autoscaler, however, and the constant tuning of forecast parameters may cause issues, such as a drop in prediction accuracy and under-fitting/over-fitting. Thus, a balance needs to be struck in the actions to be taken on SLA constraints being violated, and additional cooldowns need to be implemented on the frequency of such actions being implemented.

32.11 Summary and Conclusions

Auto-scaling on edge computing architectures remains a major research area, due to the resource and time constraints involved. In this chapter, we describe edge computing architecture, how it differs from general cloud computing paradigms, and how these architectures may be implemented using Kubernetes. Furthermore, we describe service level agreements and how they are implemented in edge computing. We have also shown that several reactive, proactive, and hybrid autoscalers exist for cloud computing environments. However, these algorithms do not always work well in edge computing scenarios, due to a variety of reasons. Each of these algorithms and their corresponding limitations are presented in this chapter. Finally, we review the future areas of research, such as multivariate solutions that may improve the performance of such algorithms and encourage their adoption industry-wide.

Acknowledgments We thank the editors and reviewers for their insightful comments and suggestions on improving this chapter.

Competing Interest Declaration The author(s) has no competing interests to declare that are relevant to the content of this manuscript.

References

1. Rimal, B., Choi, E., Lumb, I.: A taxonomy and survey of cloud computing systems. In: 2009 Fifth International Joint Conference On INC, IMS And IDC, pp. 44–51 (2009)
2. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE Internet Things J. **3**, 637–646 (2016)
3. Satyanarayanan, M.: The emergence of edge computing. Computer **50**, 30–39 (2017)
4. Cao, K., Liu, Y., Meng, G., Sun, Q.: An overview on edge computing research. IEEE Access **8**, 85714–85728 (2020)
5. Liu, F., Tang, G., Li, Y., Cai, Z., Zhang, X., Zhou, T.: A survey on edge computing systems and tools. Proc. IEEE **107**, 1537–1562 (2019)
6. Ren, J., Zhang, D., He, S., Zhang, Y., Li, T.: A survey on end-edge-cloud orchestrated network computing paradigms: transparent computing, mobile edge computing, fog computing, and cloudlet. ACM Comput. Surv. (CSUR) **52**, 1–36 (2019)
7. Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P., Nikolopoulos, D.: Challenges and opportunities in edge computing. In: 2016 IEEE International Conference On Smart Cloud (SmartCloud), pp. 20–26 (2016)
8. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10ccc), pp. 583–590 (2015)
9. Podolskiy, V., Jindal, A., Gerndt, M.: IAAS reactive autoscaling performance challenges. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 954–957 (2018)
10. Straesser, M., Grohmann, J., Kistowski, J., Eismann, S., Bauer, A., Kounev, S.: Why is it not solved yet? challenges for production-ready autoscaling. In: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, pp. 105–115 (2022)
11. Patel, P., Ranabahu, A., Sheth, A.: Service level agreement in cloud computing (2009)
12. Mirobi, G., Arockiam, L.: Service level agreement in cloud computing: An overview. In: 2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), pp. 753–758 (2015)
13. Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., Babar, M.: Understanding and addressing quality attributes of microservices architecture: a Systematic literature review. Inf. Softw. Technol. **131**, 106449 (2021)
14. Nabi, M., Toeroe, M., Khendek, F.: Availability in the cloud: state of the art. J. Netw. Comput. Appl. **60**, 54–67 (2016)
15. Casalicchio, E.: Container orchestration: a survey. In: Systems Modeling: Methodologies And Tools, pp. 221–235 (2019)
16. Vayghan, L., Saied, M., Toeroe, M., Khendek, F.: A Kubernetes controller for managing the availability of elastic microservice based stateful applications. J. Syst. Softw. **175**, 110924 (2021)
17. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**, 50–57 (2016)
18. Baier, J.: Getting Started with Kubernetes. Packt Publishing Ltd, Birmingham (2017)
19. Baresi, L., Hu, D., Quattrocchi, G., Terracciano, L.: KOSMOS: vertical and horizontal resource autoscaling for kubernetes. In: International Conference on Service-Oriented Computing, pp. 821–829 (2021)
20. Yu, W., Liang, F., He, X., Hatcher, W., Lu, C., Lin, J., Yang, X.: A survey on the edge computing for the Internet of Things. IEEE Access **6**, 6900–6919 (2017)
21. Smeliansky, R.: Hierarchical edge computing. In: 2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC), pp. 1–11 (2018)
22. Tong, L., Li, Y., Gao, W.: A hierarchical edge cloud architecture for mobile computing. In: IEEE INFOCOM 2016-The 35th Annual IEEE International Conference On Computer Communications, pp. 1–9 (2016)
23. Jararweh, Y., Doulat, A., AlQudah, O., Ahmed, E., Al-Ayyoub, M., Benkhelifa, E.: The future of mobile cloud computing: integrating cloudlets and mobile edge computing. In: 2016 23rd International Conference on Telecommunications (ICT), pp. 1–5 (2016)
24. Wang, G., Zhao, Y., Huang, J., Wang, W.: The controller placement problem in software defined networking: a survey. IEEE Netw. **31**, 21–27 (2017)
25. Liu, J., Zhang, S., Kato, N., Ujikawa, H., Suzuki, K.: Device-to-device communications for enhancing quality of experience in software defined multi-tier LTE-A networks. IEEE Netw. **29**, 46–52 (2015)
26. Du, P., Nakao, A.: Application specific mobile edge computing through network softwarization. In: 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), pp. 130–135 (2016)
27. Jararweh, Y., Doulat, A., Darabseh, A., Alsmirat, M., Al-Ayyoub, M., Benkhelifa, E.: SDMEC: software defined system for mobile edge computing. In: 2016 IEEE International Conference On Cloud Engineering Workshop (IC2EW), pp. 88–93 (2016)
28. Salman, O., Elhajj, I., Kayssi, A., Chehab, A.: Edge computing enabling the Internet of Things. In: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), pp. 603–608 (2015)
29. Premsankar, G., Di Francesco, M., Taleb, T.: Edge computing for the Internet of Things: a case study. IEEE Internet Things J. **5**, 1275–1284 (2018)
30. Mishra, T., Wang, M., Metwally, A., Bogu, G., Brooks, A., Bahmani, A., Alavi, A., Celli, A., Higgs, E., Dagan-Rosenfeld, O., et al.: Early detection of COVID-19 using a smartwatch. In: MedRxiv, pp. 2020–07 (2020)

31. Kristiani, E., Yang, C., Wang, Y., Huang, C.: Implementation of an edge computing architecture using OpenStack and Kubernetes. In: Information Science and Applications 2018, pp. 675–685 (2019)
32. Phan, L., Kim, T., et al.: Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure. IEEE Access **10**, 18966–18977 (2022)
33. Beni, E., Truyen, E., Lagaisse, B., Joosen, W., Dieltjens, J.: Reducing cold starts during elastic scaling of containers in kubernetes. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing, pp. 60–68 (2021)
34. Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., Appavoo, J.: SEUSS: rapid serverless deployment using environment snapshots. arXiv Preprint ArXiv:1910.01558 (2019)
35. Lorenzo, C., Guillaume, P., Bellavista, P.: FogDocker: start container now fetch image later. In: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC) (2019)
36. Lin, P., Glikson, A.: Mitigating cold starts in serverless platforms: a pool-based approach. arXiv Preprint ArXiv:1903.12221 (2019)
37. Venticinque, S., Aversa, R., Di Martino, B., Rak, M., Petcu, D.: A cloud agency for SLA negotiation and management. In: Euro-Par 2010 Parallel Processing Workshops: HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, August 31–September 3, 2010, Revised Selected Papers 16, pp. 587–594 (2011)
38. Sakr, S., Liu, A.: SLA-based and consumer-centric dynamic provisioning for cloud databases. In: 2012 IEEE Fifth International Conference on Cloud Computing, pp. 360–367 (2012)
39. Houlihan, R., Du, X., Tan, C., Wu, J., Guizani, M.: Auditing cloud service level agreement on VM CPU speed. In: 2014 IEEE International Conference on Communications (ICC), pp. 799–803 (2014)
40. Wu, L., Garg, S., Versteeg, S., Buyya, R.: SLA-based resource provisioning for hosted software-as-a-service applications in cloud computing environments. IEEE Trans. Serv. Comput. **7**, 465–485 (2013)
41. Rajavel, R., Mala, T.: Achieving service level agreement in cloud environment using job prioritization in hierarchical scheduling. In: Proceedings of the International Conference On Information Systems Design and Intelligent Applications 2012 (INDIA 2012) Held in Visakhapatnam, India, January 2012, pp. 547–554 (2012)
42. Nunes, J., Bianchi, T., Iwasaki, A., Nakagawa, E.: State of the art on microservices autoscaling: An overview. In: Anais Do XLVIII Seminário Integrado De Software E Hardware, pp. 30–38 (2021)
43. Dogani, J., Namvar, R., Khunjush, F.: Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. In: Computer Communications (2023)
44. Kampars, J., Pinka, K.: Auto-scaling and adjustment platform for cloud-based systems. In: Environment. Technologies. Resources. Proceedings of the International Scientific And Practical Conference, vol. 2, pp. 52–57 (2017)
45. Zhang, F., Tang, X., Li, X., Khan, S., Li, Z.: Quantifying cloud elasticity with container-based autoscaling. Futur. Gener. Comput. Syst. **98**, 672–681 (2019)
46. Srirama, S., Adhikari, M., Paul, S.: Application deployment using containers with auto-scaling for microservices in cloud environment. J. Netw. Comput. Appl. **160**, 102629 (2020)
47. Hoenisch, P., Weber, I., Schulte, S., Zhu, L., Fekete, A.: Four-fold auto-scaling on a contemporary deployment platform using docker containers. In: Proceedings of the Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16–19, 2015, pp. 316–323 (2015)
48. Santos, G., Paulino, H., Vardasca, T. (2020). QoE-aware auto-scaling of heterogeneous containerized services (and its application to health services). In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, pp. 242–249
49. Sheganaku, G., Schulte, S., Waibel, P., Weber, I.: Cost-efficient auto-scaling of container-based elastic processes. Futur. Gener. Comput. Syst. **138**, 296–312 (2023)
50. Taherizadeh, S., Stankovski, V.: Dynamic multi-level auto-scaling rules for containerized applications. Comput. J. **62**, 174–197 (2019)
51. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. **12**, 559–592 (2014)
52. Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. In: International Conference on Machine Learning, pp. 1310–1318 (2013)
53. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press In, New York (2001)
54. Ju, L., Singh, P., Toor, S.: Proactive autoscaling for edge computing systems with kubernetes. In: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, pp. 1–8 (2021)
55. Meng, Y., Rao, R., Zhang, X., Hong, P.: CRUPA: a container resource utilization prediction algorithm for auto-scaling based on time series analysis. In: 2016 International Conference on Progress in Informatics and Computing (PIC), pp. 468–472 (2016)
56. Imdoukh, M., Ahmad, I., Alfailakawi, M.: Machine learning-based auto-scaling for containerized applications. Neural Comput. Appl. **32**, 9745–9760 (2020)
57. Messias, V., Estrella, J., Ehlers, R., Santana, M., Santana, R., Reiff-Marganiec, S.: Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. Neural Comput. Appl. **27**, 2383–2406 (2016)
58. Lambora, A., Gupta, K., Chopra, K.: Genetic algorithm-A literature review. In: 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), pp. 380–384 (2019)
59. Abdullah, M., Iqbal, W., Berral, J., Polo, J., Carrera, D.: Burst-aware predictive autoscaling for containerized microservices. IEEE Trans. Serv. Comput. **15**, 1448–1460 (2020)
60. Alidoost Alanagh, Y., Firouzi, M., Rasouli Kenari, A., Shamsi, M.: Introducing an adaptive model for auto-scaling cloud computing based on workload classification. Concurrency Comput. Pract. Exper. **35**, e7720 (2023)
61. Qu, C., Calheiros, R. & Buyya, R. Auto-scaling web applications in clouds: a taxonomy and survey. ACM Comput. Surv. (CSUR) **51**, 1–33 (2018)
62. Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: On the use of fuzzy modeling in virtualized data center management. In: Fourth International Conference on Autonomic Computing (ICAC'07), pp. 25–25 (2007)
63. Lama, P., Zhou, X.: Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters. In: 2009 17th International Workshop on Quality Of Service, pp. 1–9 (2009)
64. Rampérez, V., Soriano, J., Lizcano, D., Lara, J.: FLAS: a combination of proactive and reactive auto-scaling architecture for distributed services. Futur. Gener. Comput. Syst. **118**, 56–72 (2021)
65. Biswas, A., Majumdar, S., Nandy, B., El-Haraki, A.: A hybrid auto-scaling technique for clouds processing applications with service level agreements. J. Cloud Comput. **6**, 1–22 (2017)
66. Singh, P., Kaur, A., Gupta, P., Gill, S., Jyoti, K.: RHAS: robust hybrid auto-scaling for web applications in cloud computing. Clust. Comput. **24**, 717–737 (2021)
67. Singh, P., Gupta, P., Jyoti, K.: TASM: technocrat ARIMA and SVR model for workload prediction of web applications in cloud. Clust. Comput. **22**, 619–633 (2019)



Suhrid Gupta currently works as a research assistant at the University of Melbourne. He received his Master of Computer Science degree from the University of Melbourne, with his primary research interests in the field of edge computing, auto-scaling algorithms, and machine learning.



Muhammed Tawfiqul Islam is a Lecturer in the School of Computing and Information Systems (CIS) at the University of Melbourne, Australia. He completed his PhD from the School of Computing and Information Systems (CIS) at the University of Melbourne in 2021. His research interests include resource management, cloud computing, big data, stream computing, and software-defined networks.



Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University commercializing its innovations in Cloud Computing. He has authored over 750 publications and four textbooks. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 169 with 152,400+ citations). He is among the world's top 2 most influential scientists in distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of the Scopus citation database. He served as the founding Editor-in-Chief (EiC) of IEEE Transactions on Cloud Computing and is now serving as EiC of the Journal of Software: Practice and Experience.