A Hybrid Reactive-Proactive Auto-scaling Algorithm for SLA-Constrained Edge Computing

Suhrid Gupta
University of Melbourne
Melbourne, Victoria, Australia
suhrid.gupta1@unimelb.edu.au

Muhammed Tawfiqul Islam
University of Melbourne
Melbourne, Victoria, Australia
tawfiqul.islam@unimelb.edu.au

Rajkumar Buyya
University of Melbourne
Melbourne, Victoria, Australia
rbuyya@unimelb.edu.au

Abstract

Edge computing decentralizes computing resources, allowing for novel applications in domains such as the Internet of Things (IoT) in healthcare and agriculture by reducing latency and improving performance. This decentralization is achieved through the implementation of microservice architectures, which require low latencies to meet stringent service level agreements (SLA) 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. This is achieved by sophisticated orchestration strategies such as Kubernetes, which help facilitate resource management. The orchestration strategies alone do not guarantee SLA adherence due to the inherent delay of scaling resources. Existing auto-scaling algorithms have been proposed to address these challenges, but they suffer from performance issues and configuration complexity. In this paper, a novel auto-scaling algorithm is proposed for SLA-constrained edge computing applications. This approach combines a Machine Learning (ML) based proactive auto-scaling algorithm, capable of predicting incoming resource requests to forecast demand, with a reactive autoscaler which considers current resource utilization and SLA constraints for immediate adjustments. The algorithm is integrated into Kubernetes as an extension and its performance is evaluated through extensive experiments in an edge environment with real applications. The results demonstrate that existing solutions have an SLA violation rate of up to 23%, whereas the proposed hybrid solution outperforms the baselines with an SLA violation rate of only 6%, ensuring stable SLA compliance across various applications.

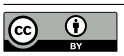
CCS Concepts

• **Computer systems organization** → **Cloud computing**; *Distributed architectures.*

Keywords

Edge Computing, Auto-scaling, Kubernetes, LSTM, SLA Compliance, Microservices

ACM Reference Format:
Suhrid Gupta, Muhammed Tawfiqul Islam, and Rajkumar Buyya. 2026. A Hybrid Reactive-Proactive Auto-scaling Algorithm for SLA-Constrained Edge Computing. In *2026 Australasian Computer Science Week (ACSW 2026), February 09–12, 2026, Melbourne, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3793811.3793826



This work is licensed under a Creative Commons Attribution 4.0 International License.
ACSW 2026, Melbourne, Australia
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2315-5/26/02
https://doi.org/10.1145/3793811.3793826

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 [21]. In the early days, a singular end-point would be used to access these services, however nowadays the architecture is a multi-regional model allowing effortless access from across the world. The increasing popularity of hand-held 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 [25] This was resolved using edge computing architectures.

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 [11]. This reduces the latency in terms of client-server communication and data processing. Cloudlets can also be easily scaled depending on the resource requirements per edge architecture. However, due to the dynamic resource requirements which 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 [30].

One method of mitigating this scaling latency is through the use of microservice applications. By employing a microservice architecture, the resources in a cloudlet are distributed as a collection of smaller deployments that are both independent and loosely coupled [31]. 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 microservice 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. Applications such as the ones used in healthcare require stringent compliance to service level agreements (SLA) 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 microservice resource allocation 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 [18].

Proactive auto-scaling solutions attempt to model resource allocation over time and effectively predict the resource requirements. By doing so, the microservice 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 [28].

To tackle these challenges, this paper proposes a hybrid approach that combines the simplicity of using reactive autoscalers, while maintaining the resource availability benefits of the proactive autoscalers. The algorithm involves a smaller-scale LSTM machine learning model which can be quickly trained to recognize the key features of the resource workload over a course of time. The autoscaler then uses the predictions from the LSTM model to scale its resources in advance. A reactive autoscaler is used to maintain the current resource requirements as the predictive model cannot provide fine-tuned predictions. The accuracy of the prediction model is gauged by monitoring the SLA metrics. If it is observed that the predictive model is performing poorly, the training parameters are automatically tuned for the next training iteration.

The **contributions** of this paper are as follows:

- Propose a hybrid auto-scaling method that mitigates the challenges present in reactive and proactive methods.
- The algorithm is implemented as an extension to Kubernetes.
- Deploy this algorithm in a real edge cluster prototype.
- Conduct extensive experiments using realistic daily workloads on a prototype microservice application.
- Compare the SLA violations of cutting-edge reactive, proactive, and default container orchestration autoscalers with the proposed hybrid algorithm.

The remainder of the paper is organized as follows. Section 2 discusses related autoscaling strategies, their benefits, and drawbacks. Section 3 formulates the problem we are attempting to solve, including SLA and the cold start problem. Section 4 provides an overview of the new hybrid autoscaler architecture and algorithm along with an analysis of its computational complexity. Section 5 deals with the experiments conducted to validate the efficacy of the proposed architecture, with Section 6 discussing the performance of the architecture. Finally, Section 7 concludes and summarizes the findings and discusses some of the future enhancements that could be made to the architecture.

2 Related Work

2.1 Reactive Autoscaling Strategies

Nunes et al. [15] 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 a reliance on predetermined resource thresholds and a delay in resource scaling, have been popular in research articles. Dogani et al. [2] stated that this was due to the simplicity and user-friendliness in developing them.

Kampars and Pinka [7] proposed a reactive auto-scaling algorithm for edge architectures based on open-source technologies. The algorithm scales in a non-standard approach, considering real-time adjustments in the application logic to determine the strategy of scaling. Zhang et al. [33] presented an algorithm for determining edge elasticity through container-based auto-scaling, demonstrating that balancing system stability with decent elasticity required careful tuning of parameters such as cooldown periods. However the lack of addressing the cold start problem results in a delay in scaling resources, violating SLA-compliance.

Phan et al. [17] proposed a reactive auto-scaling solution for edge deployments for IoT devices which dynamically allocates resources based on incoming traffic. This traffic-aware horizontal pod autoscaler (THPA) operates on top of the underlying Kubernetes architecture. 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 modelling the resource requests per Kubernetes nodes and intelligently allocating pods to the nodes with higher number of requests. The authors demonstrated that this approach provided a 150% improvement in response time and throughput. However the algorithm is not SLA compliant due to the delay in scaling resources in a reactive manner.

2.2 Proactive Autoscaling Strategies

Lorido et al. [12] showed that compared to reactive algorithms, proactive algorithms achieved better resource allocation once they had been carefully optimized. Machine learning 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.

Ju et al. [6] presented a proactive horizontal pod auto-scaling 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. However, such customizability leads to a complex deployment and hyper-parameter tuning process. This, along with a lack of initial training data causes erroneous predictions before the model corrects itself.

Meng et al. [13] created a proactive auto-scaling algorithm for forecasting the Kubernetes CPU usage of containers using a time-series prediction with ARIMA. The authors demonstrated that such an architecture reduced the forecast errors to 6.5%, as compared to the baseline of 17%. However the cost of training this model was prohibitively high, making it unsuitable for edge deployments.

Imdoukh et al. [5] 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 demonstrated that their algorithm was as accurate as existing ARIMA-based proactive solutions, but significantly reduced the prediction time, as well

Table 1: Summary of hybrid auto-scaling solutions

Features	Hybrid Algorithms					Proposed
	[32]	[10]	[20]	[1]	[27]	
Simple deployment	✓	✓	✓	✓	✓	✓
Simple parameter tuning	✓	✓	✓	✗	✗	✓
Custom metrics	✓	✗	✗	✓	✓	✓
Light-weight deployment	✓	✗	✓	✗	✗	✓
Edge architecture compliant	✗	✗	✗	✗	✗	✓
SLA-compliant	✗	✗	✓	✓	✓	✓
Minimizes deployment cost	✗	✗	✗	✗	✓	✓

as computed the minimum resource allocation required to handle future workload. However, this algorithm also suffers from the problems related to a lack of initial training data.

2.3 Hybrid Autoscaling Strategies

All the reactive and proactive approaches have their benefits and drawbacks. Thus, hybrid solutions which merge multiple autoscaling methods were proposed [19]. While hybrid algorithms for cloud-based deployments exist, integrating them into edge architectures pose 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. Table 1 shows an overview of the existing proposals compared with our solution.

In 2007, one of the first hybrid algorithms for a distributed deployment was proposed by Jing et al. [32]. This algorithm combined rule-based fuzzy inference with machine learning forecasting for dynamic resource allocation. Based on this work, Lama and Zhou [10] proposed a resource provisioning algorithm for multi-cluster setups using a hybrid autoscaler comprising of a combination of fixed fuzzy rule-based logic and a self adaptive algorithm which dynamically tuned the scaling factor.

Rampérez et al. [20] proposed a hybrid approach called Forecasted Load Auto-scaling (FLAS), which combines a predictive model for forecasting time-series resources, while the reactive model estimates other high-level metrics. The linear regression forecaster was however too simplistic to predict complex time-series. Biswas et al. [1] presented a hybrid algorithm designed for cloud computing deployments with service level agreements using an SVM model. Such an SVM-based model is expensive to train however, making it infeasible to deploy on edge deployments.

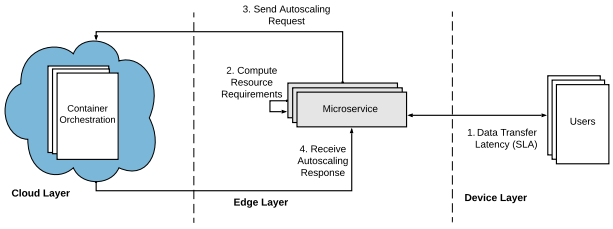
Singh et al. [27] proposed another cloud computing based autoscaler with SLA-constraints. The robust hybrid autoscaler (RHAS) was designed particularly for web applications using a modification of the ARIMA machine learning model (TASM). The technique was demonstrated to reduce cloud deployment cost and SLA violations. However, the TASM forecaster was too complex and resource intensive to be deployed on scarce resource paradigms such as the edge layer. This resource intensiveness increased the training times drastically, making it infeasible for conforming to SLA constraints on edge architectures.

The RHAS algorithm by Singh et al. [27] provided the best approach template for creating a hybrid autoscaler. Therefore, in implementing the autoscaler presented in this paper, we extended the generalized architecture of RHAS to streamline both the reactive and proactive autoscalers, while choosing a more efficient and cost-effective forecasting model to make it SLA-compliant on edge architectures, and eliminating the costly and time-consuming hyper-parameter tuning process.

3 Problem Formulation

Figure 1 shows the edge architecture layout. The cloud layer is similar to cloud-computing paradigms, wherein it manages the entire network architecture and stores large scale data. The edge layer consists of smaller scale user data storage and communication with user devices. Finally, the device layer consists of all the user devices that will interact with the edge architecture.

Figure 1: Autoscaling problem overview



The cloud layer has the most amount of resources allocated to it, which it requires when managing the entire network, computing the intensive processing of large-scale data, and coordinating the resource allocation of the edge layer. Only system-critical applications such as the controller orchestration control plane are deployed on this layer. The edge layer has far fewer resources than the cloud layer, but its proximity to the users results in lower network latency, making it ideal for resource scaling. For this reason, the edge layer consists of the orchestration tool's worker nodes and the microservice which receives and serves user data. These worker nodes allocate resources to the microservice deployments dynamically according to user requirements through the process known as auto-scaling.

3.1 SLA Constraint Definition

Cloud deployments provide several Quality of Service (QoS) metrics when considering SLA negotiations [24]. These can be broadly classified into performance metrics (response time, throughput), availability metrics (abandon rate, use rate), reliability metrics (mean failure time, mean recovery time), and cost metrics (financial and energy costs).

For this research, we utilize the performance metric *response time* of the user requests as the SLA constraint metric. This metric was chosen due to it being affected the most by intelligently auto-scaling cloud services. By using this metric, the cloud deployment guaranteed that all requests would be served under a certain threshold.

For real-time applications, the auto-scaling should adhere to the SLA metric as much as possible, and try to minimize the number of violations. An SLA constraint $\mathcal{S}_c(t)$ is defined as a metric value not exceeding above a threshold Δ agreed by both the cloud provider

and the customer:

$$\mathcal{S}_c(t) > \Delta \tag{1}$$

Following Hussain et al. [4], we consider multiple SLA thresholds for different use cases: *Flexible* (highest threshold, typical IoT applications), *Moderate* (trade-off for real-time capabilities), and *Strict* (lowest threshold for time-critical applications such as medical surgeries).

3.2 Cold-Start Problem

The auto-scaling will use a resource metric to scale resources up or down. A problem arises in the time it takes to scale these resources. This time to increase the number of resource replicas $\mathcal{R}$, which we define as the cold start time $C(t)$:

$$C(t) = \mathcal{R}_{deploy}(t) + \mathcal{R}_{register}(t) \tag{2}$$

where the cold start time is the summation of the time taken for the replica to be deployed on the data plane and registered with the control plane. The replica image download time is typically a one-time delay due to optimizations done on modern container orchestration software and can be ignored for SLA latency calculations.

When computing the SLA constraint value for a latency metric, the SLA latency can be written as the sum of the cold-start time and the round-trip time taken for the request:

$$\mathcal{S}_c(t) = C(t) + \mathcal{K}(t) + \frac{\mathcal{U}(t)}{\sum_i p_i} \tag{3}$$

where $\mathcal{K}(t)$ is the constant network latency, $\mathcal{U}(t)$ is the maximum latency of a unitary resource deployment, and $\sum_i p_i$ is the total pod count in deployment $\mathcal{D}$. The number of SLA violations $\mathcal{V} \propto C(t)$ due to the correlation between cold-start delay and the lack of available resources [16].

3.3 Optimization Problem

From Equation 3, it is clear that $\lim_{\sum_i p_i \to \infty} \frac{\mathcal{U}(t)}{\sum_i p_i} = 0$. This incentivizes ignoring intelligent auto-scaling and simply allocating maximum pods. However, most cloud providers allocate a cost for each resource assignment:

$$cost = \alpha \times \sum_i p_i \tag{4}$$

where α is the unitary resource cost. The auto-scaling optimization problem $\mathcal{P}$ can be formed:

$$\mathcal{P} = x \times \mathcal{S}_c(t) + y \times cost \tag{5}$$

The objective is to minimize both latency and cost. For this research, we configure $x = y = 0.5$, implying both are equally important. Maximizing resources in $\mathcal{D}$ while limiting cost below a threshold to reduce SLA constraint metric is akin to the famous Knapsack Problem [8]. This problem is proven to be NP-Hard, and as such no known algorithm can determine the best value in polynomial time. However, an approximation close to this best value can be computed in polynomial time through reactive rule-based or proactive machine-learning techniques.

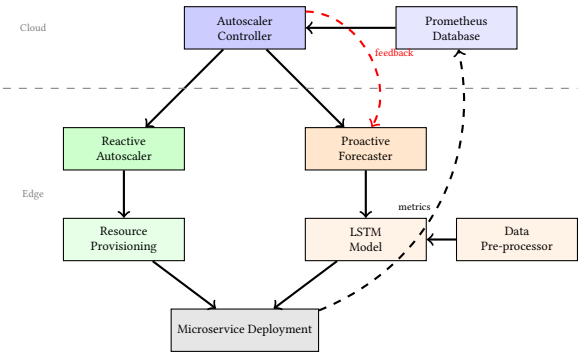


Figure 2: Proposed hybrid architecture overview

4 Proposed Hybrid Autoscaler

4.1 Architecture Overview

An overview of the hybrid autoscaler architecture is shown in Figure 2. The overall architecture is formulated using a hierarchical model. The edge node consists of three main sections. The first is the reactive auto-scaling subsystem, which has the resource provisioning module, and the configuration which dictates the cooldown logic for scaling up and down. As Zhang et al. [33] demonstrated, the microservice system stability is directly related to the careful selection of cool-down parameters.

The second subsystem is the proactive autoscaler. From a high-level perspective, there are three main components. The resource provisioning module is similar to that of the reactive autoscaler, however, it also consists of a forecaster using a deep-learning-based machine learning model, and a data pre-processing algorithm. The data pre-processing algorithm removes any noise present in the time series data, and smoothens the data curves, making it easier for the forecaster to make predictions in a low-cost manner.

Finally, the auto-scaling controller determines which auto-scaling logic will be applied to the replicas, and also keeps track of any SLA violations. It hosts the time-series metric data and has a feedback loop with the proactive autoscaler. If it detects SLA violations during a configured time window, it automatically adjusts the hyper-parameters of the proactive forecaster. Such a heuristic method eliminates the complex hyper-parameter tuning process seen in most proactive models.

4.2 Scheduler Algorithm

At a high level, a container orchestration's default horizontal pod autoscaler operates on the ratio between the current and desired metric values:

$$replicas_{desired} = \lceil replicas_{current} \times \frac{metric_{current}}{metric_{desired}} \rceil \tag{6}$$

The autoscaler controller consists of a scheduling logic module which handles when to switch between proactive and reactive auto-scaling. Algorithm 1 explains this logic. The autoscaler computes two replica values, one for the proactive forecaster which determines the replicas after $\mathcal{T}$ seconds, and one for the reactive forecaster for current requirements. If the forecasted requirement is higher than current, the scheduler outputs the forecaster replica count as desired replicas. Otherwise, the reactive replica count is used.

Algorithm 1 Hybrid Scheduler Algorithm

Input: $replicas_{current}, metric_{current}, metric_{desired}, metric_{forecast}$
Output: $replicas_{desired}$

$replicas_{forecast} \leftarrow \lceil replicas_{current} \times \frac{metric_{forecast}}{metric_{desired}} \rceil$
$replicas_{reactive} \leftarrow \lceil replicas_{current} \times \frac{metric_{current}}{metric_{desired}} \rceil$
if $replicas_{forecast} > replicas_{reactive}$ **then**
    $replicas_{desired} \leftarrow replicas_{forecast}$
else
    $replicas_{desired} \leftarrow replicas_{reactive}$
end if
return $replicas_{desired}$

4.3 Reactive Resource Provisioning

The reactive autoscaler subsystem is responsible for determining whether auto-scaling should proceed based on given configuration. The reactive algorithm's resource provisioning is built on top of the default horizontal pod autoscaler deployed by Kubernetes. The autoscaler is modified such that it has cooldown parameters set to a moderate value to ensure adaptability to SLA-constrained scenarios while maintaining system stability.

There are three important parameters key to controlling horizontal pod scaling: "tolerance", "scale up cooldown", and "scale down cooldown". The tolerance informs the autoscaler when to skip calculating new replicas:

$$tolerance = \left| \frac{metric_{desired} - metric_{current}}{metric_{desired}} \right| \tag{7}$$

By default, if tolerance is below 0.1, autoscaling is skipped. The scale-up and scale-down cooldowns control how quickly auto-scaling occurs. For the proposed autoscaler, both cooldowns are modified to 15 seconds to ensure moderate cooldown values for best system stability and SLA compliance.

4.4 Data Pre-Processor

To speed up the forecast process and reduce resource requirements, the time-series data is pre-processed to smoothen it. This makes it easier for the deep learning model to extract patterns, reduces training and validation loss, and reduces the length of the training window data sequence the LSTM requires. The smoothing is achieved using the Savitzky-Golay filter [22], which takes N points in a given time-series, with a filter width w, and calculates a polynomial average of order o [23]. The resulting data has considerably less deviations between consecutive points and is devoid of noise.

4.5 Proactive Forecaster

Several time series forecaster algorithms exist, with LSTM and ARIMA being prominent ones. Siami-Namini et al. [26] demonstrated that LSTM implementations outperformed ARIMA, reducing error rates by over 80%. Furthermore, the number of training "epochs" did not need to be set to a high value; setting significantly higher values degraded performance due to over-fitting. LSTM works well due to "rolling updates" on the model—weights are only set once when deployed, then always updated on every training call.

Algorithm 2 shows the forecaster implementation. The autoscaler controller implements a control loop every $\mathcal{P}$ seconds requesting

Algorithm 2 Proactive Forecaster Algorithm

Input: $lookback \geq 0, epochs \leq 100, learning_rate \leq 1$
Output: $metric_{forecast}$

$lstm_model \leftarrow lstm.initialize()$
$time_series \leftarrow get_latest_data()$
$lstm_input \leftarrow get_input(time_series, lookback)$
$lstm_input \leftarrow preprocess_data(lstm_input)$
$new_model \leftarrow train(lstm_input, epochs, learning_rate)$
if $validation_loss(new_model) < validation_loss(lstm_model)$ **then**
    $lstm_model \leftarrow new_model$
end if
$metric_{forecast} \leftarrow lstm_model.predict(lstm_input)$
return $metric_{forecast}$

Table 2: Proactive forecaster layer configuration

Layer Details	Output Shape	Parameters
$LSTM_1$	(10, 50)	10,400
$Dropout_1$	(10, 50)	0
$LSTM_2$	(10, 50)	20,200
$Dropout_2$	(10, 50)	0
$LSTM_3$	(50)	20,200
$Dense_1$	(540)	27,540
Total		**78,340**

the latest prediction. The forecaster pre-processes data to remove noise, performs training with configured hyper-parameters, computes validation loss, and accepts this model if it has lower validation loss than previous iterations. Finally, the model predicts future metrics and returns them to the controller.

The proactive forecaster is a deep-learning model configured with multi-step forecast output. The model consists of three LSTM layers, alternated with two dropout layers (to prevent over-fitting), and a final densely connected neural-network generating the forecaster output. The output is 540 data points (approximately 24 hours of workload), so the forecaster only needs to run once daily, vastly reducing total training time.

Default hyper-parameters are: learning rate = 0.005, epochs = 75, batch size = 100, with Adam optimizer [9]. An "early-stop" function halts training if loss does not decrease for 10 consecutive epochs. The model training, validation, and error comparison takes approximately 3 minutes, after which the model predicts the subsequent day's forecast in under 10 seconds.

4.6 SLA-based Heuristic Feedback

The autoscaler controller constantly checks for SLA violations using a control loop. Typically, SLA checks are done for a sufficiently lengthy period such as one day. If an SLA violation is found, it is concluded that the application was unable to autoscale quickly enough to avoid the cold start problem. This could be due to insufficient training data or conservative hyper-parameter selections.

To temporarily boost learning, the controller decreases the learning rate (to increase probability of escaping local minima), increases batch size (to reduce under-fitting), and increases epochs (to reduce loss). All parameters have thresholds to prevent over-fitting or infeasibly lengthy training times. Algorithm 3 shows this implementation.

Algorithm 3 SLA-based Heuristic Feedback

Input: $\mathcal{V}, learning_rate, batch_size, epochs$
Output: $hyperparameters_{modified}$

$initial_rate \leftarrow learning_rate$
$initial_batch \leftarrow batch_size$
$initial_epochs \leftarrow epochs$
if $\mathcal{V} > 0$ **then**
  $batch_size \leftarrow \min(batch_size + 10, 200)$
  $learning_rate \leftarrow \max(learning_rate - 0.0005, 0.002)$
  $epochs \leftarrow \min(epochs + 5, 100)$
else
  Reset to initial values
end if
return $(learning_rate, batch_size, epochs)$

If the feedback control loop discovers no SLA-violations during a time-period, it concludes that the LSTM has sufficiently learned the primary characteristics. The "rolling-updates" feature of LSTM allows safely resetting hyper-parameters while preserving learning and weights of previous training rounds.

4.7 Complexity Analysis

Assuming the hybrid autoscaler $\mathcal{H}$ takes time-series data of length $\mathcal{N}$, stored in an array data structure, and LSTM weights as a two-dimensional matrix of size $A \times B$:

Space Complexity: The time-series array has complexity $O(N)$ and weights are $O(N^2)$. Thus:

$$Complexity_{space}(\mathcal{H}) = O(N^2) \tag{8}$$

Time Complexity: The reactive autoscaler and controller only compute tolerance values—constant operations: $O(1)$. For the proactive autoscaler, the LSTM internally computes matrix multiplications. For dimensions m and n:

$$Complexity_{time}(proactive) = O(m \times (m + n + 1)) = O(N^2) \tag{9}$$

For τ training epochs (a constant), the final complexity remains $O(N^2)$. Combining all components:

$$Complexity_{time}(\mathcal{H}) = O(1) + O(1) + O(N^2) = O(N^2) \tag{10}$$

The hybrid algorithm performs in polynomial time complexity, providing an approximation for the NP-Hard optimization problem.

5 Experimental Setup

5.1 Cluster Configuration

For the underlying virtual machine (VM) setup, servers in a private cloud were leveraged. The setup consisted of 6 VMs, using a total of 24 CPU cores and 80GB of memory. These servers were separated into a cloud and an edge layer. The servers on the cloud layer have substantially higher CPU cores and memory compared to the edge layer, to simulate resource scarcity in the edge layer. The cloud layer also contained a 200GB persistent storage volume for Prometheus data, while the edge layer stored time series in RAM. A simulated latency was added between inter-layer communication to mimic perceived distance between edge nodes and data centers.

Each server uses Ubuntu 22.04. Kubernetes v1.28.2 is used as the container orchestration technology. CRI-O was installed as the container runtime, and Flannel for inter-pod communication. A bare-metal Kubernetes implementation was used for maximum flexibility, with the control plane on the cloud layer and data plane on the edge layer.

Table 3: Cluster architectural layout

Node	Layer	CPU	Memory
Control-Plane-K8s	Cloud	8 cores	32GB
Control-Plane-DB	Cloud	8 cores	32GB
Data-Plane-1	Edge	2 cores	4GB
Data-Plane-2	Edge	2 cores	4GB
Data-Plane-3	Edge	2 cores	4GB
Data-Plane-4	Edge	2 cores	4GB

5.2 Benchmark Application

DeathStarBench [3], a social network microservice implementation, was deployed for conducting benchmarks on edge architectures with SLA constraints. The application mimics a typical large-scale social network supporting common actions: registering and login, creating user posts, reading timelines, receiving follower recommendations, following/unfollowing users, and searching.

The end-to-end service uses HTTP requests processed by NGINX load balancer, which communicates with microservices in the logic layer for composing and displaying user and timeline posts. The logic layer handles posts containing text, links, and media. Results are stored using memcached for caching and MongoDB for persistent storage.

Based on the wrk2 benchmark, two APIs were identified for testing. One was a GET call to user's home timeline (*home-timeline-service*), and the other was a POST request for creating posts (*compose-post-service*). These were identified as bottlenecks through Jaeger tracing, making them prime targets for auto-scaling.

5.3 Workload Generation

The social media deployment comes with an HTTP workload generator, wrk2, which creates realistic simulation of typical daily workload. A typical IoT application in the edge has a semi-predictable workload pattern. Tadakamalla and Menascé [29] demonstrated through a survey that IoT application workloads can be well approximated using a lognormal distribution, and daily routines of users greatly affect workload patterns.

The workload assumes peaks in morning and evening, moderate usage during afternoon, and lowest at night. The workload simulator was modified to introduce randomness to mimic realistic weekly workloads, varying on occasions such as weekends and holidays. A total of approximately 2,550,000 requests per experiment were sent to simulate a workload pattern the architecture may experience over five days. The simulation intended to keep the effects of the traffic the same, but shorten the time taken to run the experiments.

5.4 Baseline Algorithms

Three baseline algorithms were chosen for comparison, all auto-scaling at the same CPU threshold:

(1) **Default Kubernetes HPA**: No modifications—scale-up cooldown is 0 seconds, scale-down is 300 seconds. No knowledge of workload distribution or SLA violations on edge nodes.

Table 4: Experimental SLA constraints

SLA Type	GET latency (ms)	POST latency (ms)
Flexible	150	1000
Moderate	125	900
Strict	100	800

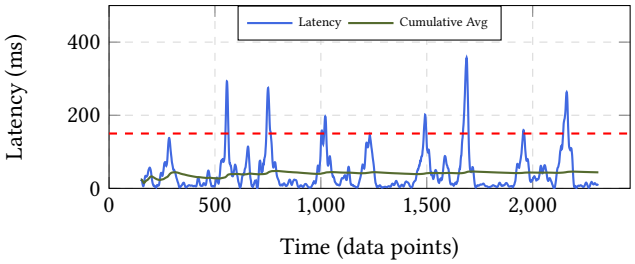


Figure 3: Default Kubernetes autoscaler latency for GET requests

(2) **Traffic Aware HPA (THPA)** [17]: Computes ratio of workloads on different edge nodes with deployment pods, scaling resources in commensurate proportion.
(3) **Proactive Pod Autoscaler (PPA)** [6]: Uses an LSTM model similar to our hybrid autoscaler but without pre-processing, dealing with more complex time-series data, requiring deeper architecture. The LSTM continuously loops through time-series data and saves forecast results. An update loop updates the model using latest forecasts. Hyper-parameters are carefully tuned but no SLA feedback is provided.

5.5 SLA Thresholds

According to Nilsson and Yngwe [14], user experience is negatively affected by higher API latency. Their research found three latency brackets: ≤100ms (instantaneous), ≤1 second (slight delay), and >10 seconds (user loses focus). Based on this, we defined three SLA categories (Table 4).

6 Performance Evaluation

Two independent experiments were conducted to validate the hybrid autoscaler performance. The social media application was first tested using GET requests to autoscale *home-timeline-service*. Then, a more demanding workload was applied using POST requests for *compose-post-service*. Both experiments used the workload generation algorithm over five days.

6.1 Request Latency Analysis

6.1.1 Default Kubernetes Autoscaler Baseline. Figure 3 shows the default Kubernetes HPA results for GET requests. The autoscaler was merely a primitive reactive implementation with no knowledge of which edge nodes experienced heavy traffic. Thus, it blindly assigned pods in a round-robin manner. Additionally, the autoscaler required significant time to register new pods, falling victim to the cold start problem. This results in significant latency spikes before resources are adjusted. The latency exceeded 300ms at some points, significantly large enough to degrade user experience. By the fifth day, cumulative average latency was nearly 50ms.

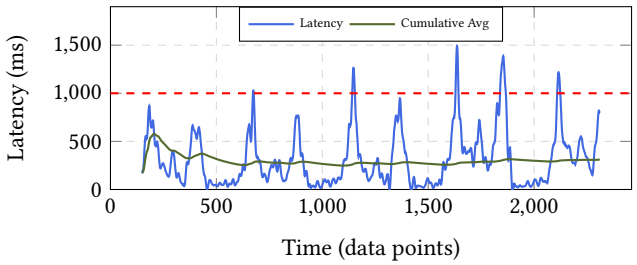


Figure 4: Default Kubernetes autoscaler latency for POST requests

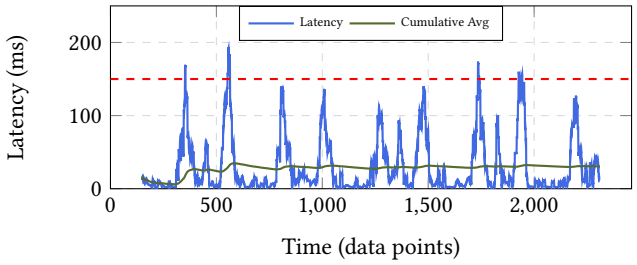


Figure 5: THPA reactive autoscaler latency for GET requests

For POST requests (Figure 4), the shortcomings were exposed even more by increased demands. During daily workload spikes, the autoscaler regularly breached 1000ms, with values peaking at almost 1450ms—more than 45% above threshold. Furthermore, the average latency hovered around 400ms throughout. Investigation revealed three issues: (1) cold start problem adding constant latency, (2) avalanche effect from resources not being available timely, causing connections to be dropped and 60-second timeouts, and (3) uneven distribution of requests due to round-robin scheduling.

6.1.2 Reactive THPA Autoscaler Baseline. Unlike the default autoscaler, THPA keeps track of which edge nodes receive significant requests and assigns pods accordingly. This resulted in significantly improved latency for GET requests (Figure 5). While still suffering from cold start, the more intelligent resource assignment resulted in fewer availability issues. However, the SLA threshold of 150ms was still regularly breached, though breaches never exceeded 200ms. The average latency was 25-30ms.

For POST requests (Figure 6), the request-aware architecture eliminated dropped request issues. The avalanche effect was somewhat mitigated. However, cold start still caused spikes above SLA threshold on multiple occasions, with latency nearly hitting 1400ms before correcting. Cumulative average latency was around 200ms—substantially lower than default.

6.1.3 Proactive PPA Autoscaler Baseline. The PPA algorithm attempts to predict workload before it is requested, eliminating cold start in ideal conditions. However, experiments showed otherwise. Because the autoscaler was purely proactive, it requires a deep LSTM model with several layers and large training epochs. This deep model took more than 50 minutes to properly train for 24-hour predictions due to edge architecture's lack of resources.

Figure 7 shows GET request results. Initially, latency continually spiked causing many SLA violations—more than the reactive autoscaler. However, after several days of training, rolling updates

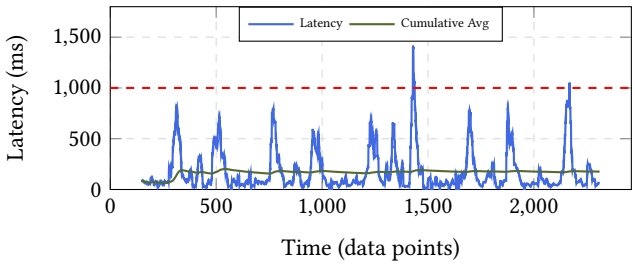


Figure 6: THPA reactive autoscaler latency for POST requests

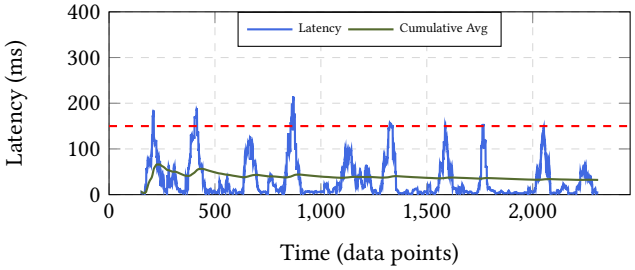


Figure 7: PPA proactive autoscaler latency for GET requests

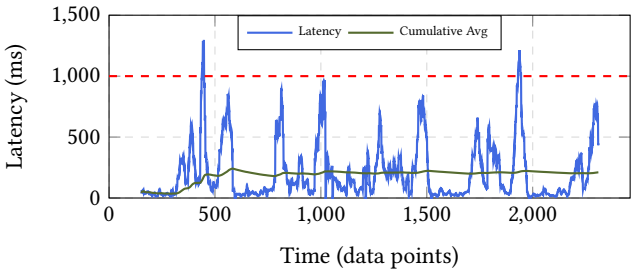


Figure 8: PPA proactive autoscaler latency for POST requests

stabilized the latency. SLA violations were not as severe as default baseline but comparatively greater than reactive, exceeding 200ms for several minutes daily. Average latency approached 50ms.

For POST requests (Figure 8), latency initially spiked above 1200ms before stabilizing, with one more spike on the last day. The first spike occurred due to insufficient training data—the complex LSTM without pre-processing made it difficult to correctly predict data curves early. This resolved as more data was added, but a threshold was reached where data was so large that forecasting took significantly longer, causing the final day spike. Average latency was around 200ms.

6.1.4 Proposed Hybrid Autoscaler. Finally, with baselines established, the hybrid algorithm was tested. This approach mitigates issues seen in both reactive and proactive approaches. The autoscaler is extremely lightweight and easy to configure since no hyper-parameter tuning is required. The proactive forecaster accurately predicts the beginning of workload spikes, eliminating cold start. The forecaster cannot accurately predict middle and end of daily workloads, but this is not an issue since the reactive algorithm handles these, ensuring SLA compliance.

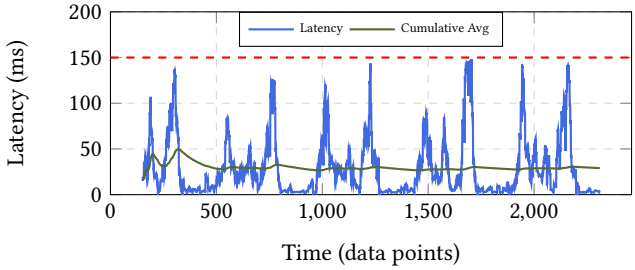


Figure 9: Hybrid autoscaler latency for GET requests

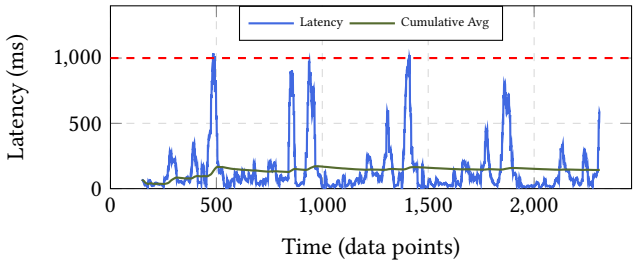


Figure 10: Hybrid autoscaler latency for POST requests

Figure 9 shows the GET request results. No SLA violations occurred during the five-day workload. The controller did not intervene in LSTM training or modify forecaster hyper-parameters. Average latency was around 30ms, similar to THPA reactive implementation. This performance was a significant improvement over baseline algorithms.

Figure 10 shows POST request results. Only one SLA violation occurred on the first day due to lack of training data causing erroneous prediction. However, the reactive subsystem took over and scaled resources accordingly, so the threshold was only breached slightly (peaking at ~1020ms). After this violation, the controller deduced that training needed kick-starting through hyper-parameter tuning. New hyper-parameter values were provided in the next training cycle, and no violations occurred the next day. The controller then reset hyper-parameters to speed up training, and even though latency approached 990ms on the third day, no further violations occurred. Average latency was below 200ms.

The hybrid approach nearly eliminated the cold start problem very early in the experiment. Initial forecasting difficulties were quickly resolved by the controller's corrective instructions. All this was done with no user intervention, making the autoscaler extremely autonomous. The algorithm completed training within a few minutes, allowing quick resource registration. CPU utilization

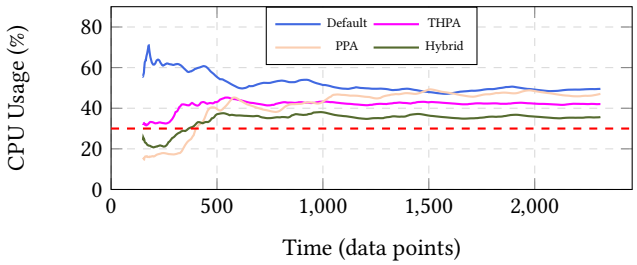


Figure 11: CPU workload distribution for POST requests

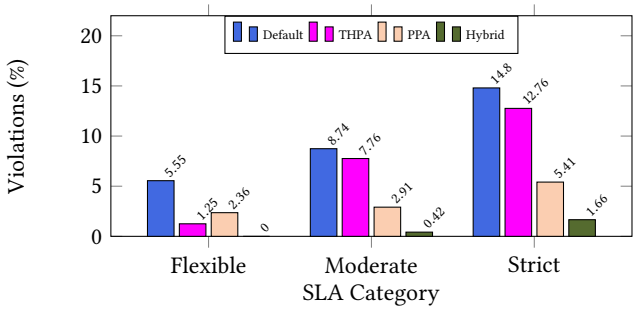


Figure 12: SLA violation rates for GET requests

never exceeded 100%, so no user requests were dropped, allowing full system availability.

6.2 CPU Workload Distribution

The distribution of CPU workload across deployment pods is another important metric. The goal is to maximize deployment resources while minimizing costs. Ideally, the distributed workload should hover at $\frac{\mathcal{A}}{2}$ where $\mathcal{A}$ is the autoscaler threshold. When workload approaches $\mathcal{A}$, not enough pods are deployed, causing queued or dropped requests. When workload tends towards 0, too many pods have been assigned, increasing costs with low latency reduction returns.

Figure 11 shows average CPU utilization for all algorithms. For GET requests (threshold 50%, ideal 25%), Default achieved ~35%, PPA ~33%, THPA ~30%, and Hybrid ~26%. For POST requests (threshold 60%, ideal 30%), Default peaked at 70% before stabilizing at 50% (with uneven distribution), PPA at ~45%, THPA at ~42%, and Hybrid at ~35%—closest to optimal in both cases.

6.3 SLA Violation Rates

As demonstrated above, the hybrid autoscaler performed significantly better than baselines with flexible SLA thresholds. For thorough demonstration, all algorithms were tested on moderate and strict thresholds. The workload was run for five days, with autoscaling performed only for the last two days to ensure best possible results regardless of training data length.

6.3.1 GET Request SLA Violations. Figure 12 and Table 5 show SLA violations for GET requests across all threshold categories.

For the flexible category, Default performed worst (5.55%), PPA achieved 2.36% (increased due to training model complexity), THPA performed well (1.25%), and Hybrid achieved **0%**—qualifying for "highly available" SLA deployment.

Table 5: SLA violation counts for GET requests

Algorithm	Flexible	Moderate	Strict
Total Requests	2,550,000	1,220,000	1,220,000
Default	141,525	106,628	180,560
THPA	31,875	94,672	155,672
PPA	60,180	35,502	66,002
Hybrid	**0**	**5,124**	**20,252**

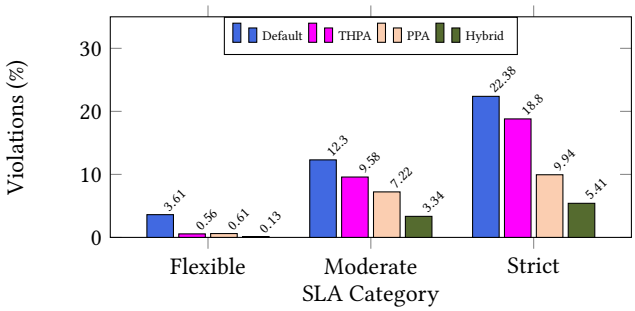


Figure 13: SLA violation rates for POST requests

Table 6: SLA violation counts for POST requests

Algorithm	Flexible	Moderate	Strict
Total Requests	2,550,000	1,220,000	1,220,000
Default	92,055	150,060	273,036
THPA	14,280	116,876	229,360
PPA	15,555	88,084	121,268
Hybrid	**3,315**	**40,748**	**66,002**

The moderate threshold proved far more difficult. Default and THPA showed similarly poor results (8.74% and 7.76% respectively), demonstrating the importance of mitigating cold start. PPA achieved 2.91%, while Hybrid still achieved best results with only 0.42% violations (initial violations quickly counteracted by hyper-parameter tuning).

For the strict threshold, Default and reactive autoscalers performed poorly (14.8% and 12.76%). PPA clearly showed cold start importance with only 5.41%. Hybrid performed substantially better with only 1.66%.

6.3.2 POST Request SLA Violations. Figure 13 and Table 6 show SLA violations for POST requests.

For the flexible category, Default was worst (3.61%, including dropped requests). Reactive and proactive performed similarly (0.56% and 0.61%)—the lenient threshold means proactive cannot display cold start mitigation benefits. Hybrid achieved 0.13%, resulting in approximately 99.9% availability (near "high availability").

The moderate threshold proved far more difficult. Default failed for 12.3% of requests. Proactive demonstrated cold start importance, achieving 7.22% vs 9.58% for reactive. Hybrid achieved only 3.34%.

For the strict threshold, Default was unable to cope (22.38%). Proactive outperformed reactive (9.94% vs 18.8%), clearly showing cold start importance. Hybrid performed significantly better with only 5.41%.

Over all experiments and thresholds, the hybrid approach served a minimum of 94.5% of requests in SLA-compliant manner (worst

case), and 100% in best case. The algorithm displayed robustness and adaptability while requiring little to no user customization.

7 Conclusions and Future Work

This paper provides a novel, lightweight, and SLA-compliant approach to autoscale resources on a microservice deployed on an edge architecture. The autoscaler architecture is constructed using open source subsystems, implementing a hybrid approach that combines reactive and proactive auto-scaling to address the cold start problem while maintaining simplicity.

The contributions include: (1) identifying major bottlenecks of auto-scaling on edge deployment compared to cloud architecture, (2) designing a novel hybrid auto-scaling architecture specifically for edge paradigms, and (3) streamlining the forecaster to run on resource-limited edge deployments cost-effectively.

The autoscaler was tested on a production-ready social network microservice deployment, and results compared with cutting-edge autoscalers. The proposed autoscaler achieved a maximum SLA violation rate of 5.41%, compared to 18.8–22.38% for state-of-the-art autoscalers. Tests further demonstrated that the autoscaler significantly reduced SLA violations while keeping deployment costs low by assigning resources to maintain utilized resources at approximately half of the configured auto-scaling threshold.

Current limitations include single-metric SLA constraints and horizontal-only scaling. Future directions include: multi-variate forecasting (CPU + memory), vertical pod auto-scaling integration, multi-SLA constraint support with weighted importance, and cluster-level auto-scaling for edge node management.

References

[1] Anshuman Biswas, Shikharesh Majumdar, Biswajit Nandy, and Ali El-Haraki. 2017. A hybrid auto-scaling technique for clouds processing applications with service level agreements. *Journal of Cloud Computing* 6 (2017), 1–22.

[2] Javad Dogani, Reza Namvar, and Farshad Khunjush. 2023. Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. *Computer Communications* (2023).

[3] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[4] Walayat Hussain, Farookh Khadeer Hussain, and Omar Khadeer Hussain. 2016. SLA management framework to avoid violation in cloud. In *Neural Information Processing: 23rd International Conference, ICONIP 2016, Kyoto, Japan, October 16–21, 2016, Proceedings, Part III 23*. Springer, 309–316.

[5] Mahmoud Imdoukh, Imtiaz Ahmad, and Mohammad Gh Alfailakawi. 2020. Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications* 32, 13 (2020), 9745–9760.

[6] Li Ju, Prashant Singh, and Salman Toor. 2021. Proactive autoscaling for edge computing systems with kubernetes. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. 1–8.

[7] Jānis Kampars and Krišjānis Pinka. 2017. Auto-scaling and adjustment platform for cloud-based systems. In *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference*, Vol. 2. 52–57.

[8] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. Introduction to NP-Completeness of knapsack problems. *Knapsack Problems* (2004), 483–493.

[9] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[10] Palden Lama and Xiaobo Zhou. 2009. Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters. In *2009 17th International Workshop on Quality of Service*. IEEE, 1–9.

[11] Fang Liu, Guoming Tang, Youhuizi Li, Zhiping Cai, Xingzhou Zhang, and Tongqing Zhou. 2019. A survey on edge computing systems and tools. *Proc. IEEE* 107, 8 (2019), 1537–1562.

[12] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing* 12 (2014), 559–592.

[13] Yang Meng, Ruonan Rao, Xin Zhang, and Pei Hong. 2016. 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)*. IEEE, 468–472.

[14] Oscar Nilsson and Noel Yngwe. 2022. API Latency and User Experience: What Aspects Impact Latency and What are the Implications for Company Performance?

[15] Joao Paulo KS Nunes, Thiago Bianchi, Anderson Y Iwasaki, and Elisa Yumi Nakagawa. 2021. State of the art on microservices autoscaling: An overview. *Anais do XLVIII Seminário Integrado de Software e Hardware* (2021), 30–38.

[16] Anjali Patel and Nisha Chaurasia. 2021. A Systematic Review of Energy Consumption and SLA Violation Conscious Adaptive Threshold based Virtual Machine Migration. In *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*. IEEE, 39–44.

[17] Linh-An Phan, Taehong Kim, et al. 2022. Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure. *IEEE Access* 10 (2022), 18966–18977.

[18] Vladimir Podolskiy, Anshul Jindal, and Michael Gerndt. 2018. IaaS reactive autoscaling performance challenges. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 954–957.

[19] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Auto-scaling web applications in clouds: A taxonomy and survey. *Comput. Surveys* 51, 4 (2018), 1–33.

[20] Víctor Rampérez, Javier Soriano, David Lizcano, and Juan A Lara. 2021. FLAS: A combination of proactive and reactive auto-scaling architecture for distributed services. *Future Generation Computer Systems* 118 (2021), 56–72.

[21] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. 2009. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC*. IEEE, 44–51.

[22] Abraham Savitzky and Marcel JE Golay. 1964. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry* 36, 8 (1964), 1627–1639.

[23] Ronald W Schafer. 2011. What is a Savitzky-Golay filter? *IEEE Signal Processing Magazine* 28, 4 (2011), 111–117.

[24] Damián Serrano, Sara Bouchenak, Yousri Kouki, Frederico Alvares de Oliveira Jr, Thomas Ledoux, Jonathan Lejeune, Julien Sopena, Luciana Arantes, and Pierre Sens. 2016. SLA guarantees for cloud services. *Future Generation Computer Systems* 54 (2016), 233–246.

[25] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[26] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. 2018. A comparison of ARIMA and LSTM in forecasting time series. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 1394–1401.

[27] Parminder Singh, Avinash Kaur, Pooja Gupta, Sukhpal Singh Gill, and Kiran Jyoti. 2021. RHAS: robust hybrid auto-scaling for web applications in cloud computing. *Cluster Computing* 24, 2 (2021), 717–737.

[28] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. 2022. Why is it not solved yet? Challenges for production-ready autoscaling. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 105–115.

[29] Uma Tadakamalla and Daniel A Menascé. 2019. Characterization of IoT workloads. In *Edge Computing–EDGE 2019: Third International Conference, Held as Part of the Services Conference Federation, SCF 2019, San Diego, CA, USA, June 25–30, 2019, Proceedings 3*. Springer, 1–15.

[30] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. 2016. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, 20–26.

[31] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 583–590.

[32] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. 2007. On the use of fuzzy modeling in virtualized data center management. In *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE, 25–25.

[33] Fan Zhang, Xuxin Tang, Xiu Li, Samee U Khan, and Zhijiang Li. 2019. Quantifying cloud elasticity with container-based autoscaling. *Future Generation Computer Systems* 98 (2019), 672–681.