Enhanced SLA Compliance in Edge Computing Applications through Hybrid Proactive-Reactive Autoscaling

by

Suhrid Gupta
Student Number: 1313675

Supervised by
Dr. Tawfiq Islam and Prof. Rajkumar Buyya

A research thesis submitted in fulfillment for the
degree of Master of Computer Science

in the
School of Computing and Information Systems
Faculty of Engineering and IT
THE UNIVERSITY OF MELBOURNE

June 2024

THE UNIVERSITY OF MELBOURNE

Abstract

School of Computing and Information Systems
Faculty of Engineering and IT

Master of Computer Science

by Suhrid Gupta
Student Number: 1313675

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 micro-service 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 thesis, a novel autoscaling algorithm was proposed for SLA-constrained edge computing applications. This approach combined a Machine Learning (ML) based proactive auto-scaling algorithm, capable of predicting incoming resource requests to forecast demand, with a reactive autoscaler which considered current resource utilization and SLA constraints for immediate adjustments. The algorithm was integrated into Kubernetes as an extension and its performance was evaluated through extensive experiments in an edge environment with real applications. The results demonstrated that existing solutions had an SLA violation rate of up to 23%, whereas the proposed hybrid solution outperformed the baselines and had an SLA violation rate of 6%, ensuring stable SLA compliance across various applications.

Declaration of Authorship

I, Suhrid Gupta, declare that this thesis titled, “Enhanced SLA Compliance in Edge Computing Applications through Hybrid Proactive-Reactive Autoscaling” and the work presented in it are my own. I confirm that:

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- this thesis did not require clearance from the University’s ethics committee.
- the thesis is approximately 24,000 words in length (excluding text in figures, tables, code listings, bibliographies, and appendices).

Signed: ________________________________

Date: June 2024

Preface and Publications

Section 2.2.1 of the thesis which provides an overview of Kubernetes was modified from the research proposal that was undertaken in partial fulfillment of the requirements of the Master of Computer Science.

All work written by myself in support of this thesis was independently reviewed by my supervisors, Dr. Tawfiq Islam and Prof. Rajkumar Buyya. Feedback was also received from my supervisors throughout the creation of the thesis and was incorporated into the final version.

Publications

Authors	Title	Venue
Suhrid Gupta, Tawfiq Islam, Rajkumar Buyya	SLA-Driven Hybrid Autoscaling for Edge Computing: Foundations, Review, and Future Directions	ACM Computing Surveys (in submission)
	A Hybrid Reactive-Proactive Autoscaling Approach for Microservice Applications in Edge Computing Environments	Journal of Systems and Software (in submission)

Acknowledgements

I wish to thank my primary supervisor, Dr. Tawfiq Islam for his generous and invaluable advice, feedback, patience, and guidance throughout the whole research process. His continuous encouragement and insights, both personal and professional, has made an immeasurably impact on me throughout my two semesters working under his supervision. I further wish to thank my co-supervisor Prof. Rajkumar Buyya. His guidance in helping me to set up milestones as well as reviewing my research experiments and thesis ensured I met my deadlines in a timely manner. I am grateful to have had them both as my supervisors, and this research thesis would not have been possible without their support.

Finally, I wish to thank my friends and family for their selfless and constant help throughout my time here at The University of Melbourne, for always motivating me, and for their endless support which has helped me complete this research project.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

In this chapter, a brief overview of the edge computing architecture paradigm, along with its uses, benefits, and challenges with respect to resource scaling are provided in Section 1.1. These challenges lead to the research gap and questions this thesis intends to answer in Section 1.2.

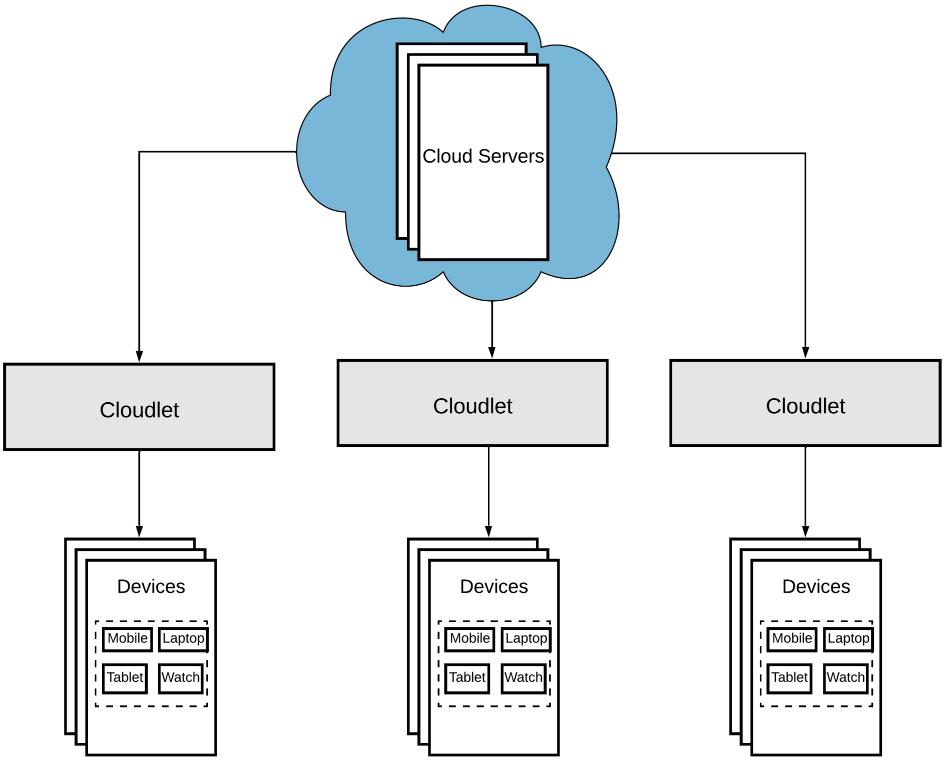
1.1 Edge Computing Overview

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 multi-regional allowing effortless access from across the world. This was achieved through the use of content delivery networks (CDN) 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 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 [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 1.1 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 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 [7].

Figure 1.1: Overview of edge computing architecture



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 (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 micro-service 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 [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].

1.2 Objectives and Thesis Structure

To tackle these challenges, this thesis 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, for example, the application latency. If it is observed that the predictive model is performing poorly, the training parameters are automatically tuned for the next training iteration.

Thus, this project aims to answer the following research questions:

- ***RQ1:*** Can we integrate reactive and proactive auto-scaling methods to develop a tailored algorithm for edge computing that eliminates the requirement to fine-tune hyper-parameters for each auto-scaling use case while being lightweight enough to be deployed and run on cloudlets?
- ***RQ2:*** Can the hybrid auto-scaling solution achieve or exceed the SLA compliance capabilities of state-of-the-art reactive and proactive auto-scaling solutions for edge computing while minimizing the cost of deploying application resources?

The thesis structure was designed based on the general guidelines provided by Gruba and Zobel [11]. In summary, the following contributions were made in this thesis:

- 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 the prototype micro-service application.
- Compare the SLA violations of the cutting-edge reactive, proactive, and default container orchestration autoscaler with the proposed hybrid algorithm for three separate threshold categories.
- Propose future work and possible extensions to the algorithm.

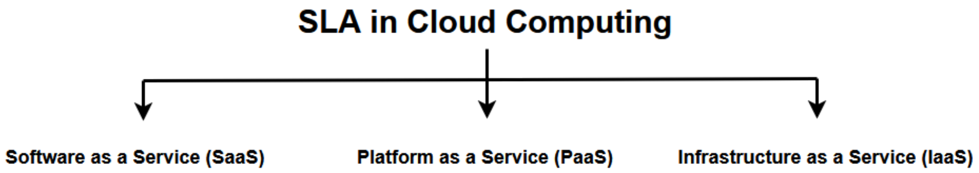
Chapter 2

Background

In this chapter, a brief introduction to Service Level Agreements is provided in Section 2.1, followed by an overview of micro-service architectures is provided in Section 2.2. This includes a brief description of the architecture of Kubernetes, along with its scheduling and auto-scaling algorithms in Section 2.3. Finally, a brief overview of time-series analysis is conducted in Section 2.4, along with some of the popular algorithms used.

2.1 Service Level Agreements

Figure 2.1: Overview of service level agreements


SLA in Cloud Computing
Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a Service (IaaS)

Cloud computing generally exposes resource using a pay-as-you-go service. These lucrative plans have led to the implementation of applications and hardwares being delivered as Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). However, consumers of such services have demands which 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 [12].

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,

Table 2.1: Summary of SLA availability

Availability %	Monthly Downtime	Yearly Downtime
90%	72 hours	36.5 days
99%	7.2 hours	3.65 days
99.9%	43.8 minutes	8.76 hours
99.99%	4.38 minutes	52.56 minutes
99.999%	25.9 seconds	5.26 minutes

one of the most common metric by which SLAs are negotiated between providers and consumers is the availability of service.

2.1.1 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 [13]. The downtime can be calculated using the formulae:

$$downtime_{monthly} = \frac{100 - Availability\%}{100} \times 30 \times 24 \tag{2.1}$$

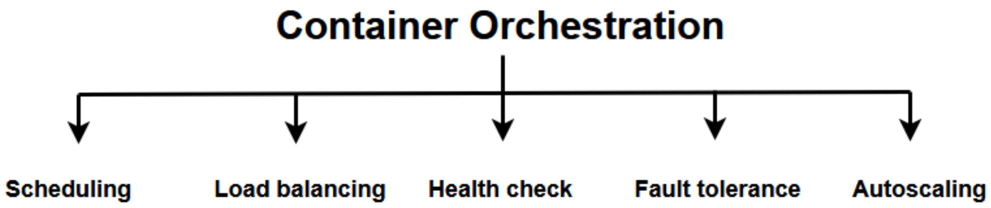
$$downtime_{yearly} = \frac{100 - Availability\%}{100} \times 365 \tag{2.2}$$

Table 2.1 shows the expected down-times for several SLA availability percentages.

2.2 Microservice 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 [14]. 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 minutes of down time per year [15]. Therefore, an orchestration mechanism is required to manage the deployment and communication of these containers.

FIGURE 2.2: Features of container orchestration



Container orchestration allows the micro-service application to customize how the deployment, monitoring, and controlling functions [16]. Figure 2.2 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 light-weight 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.

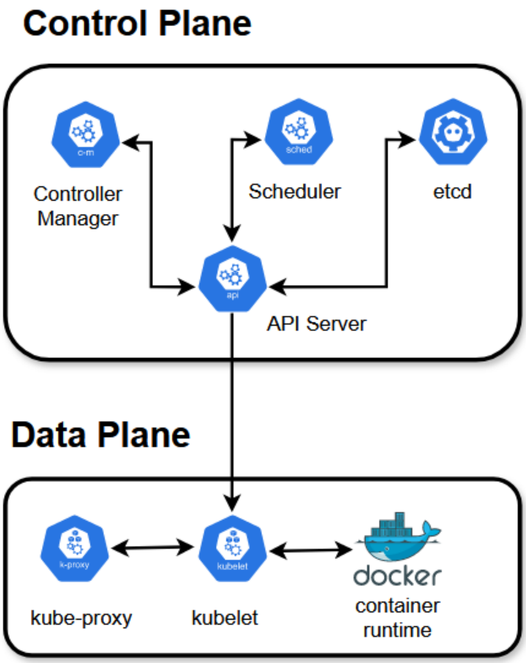
Autoscaling is the process of automatically adding or removing resources or containers. Internal metrics such as CPU usage are typically used, however custom policies can also be implemented at the discretion of the developer.

2.2.1 Kubernetes Architecture

Kubernetes [1] is one of the most popular open-source container orchestration platforms [17]. 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 [18]. Figure 2.3 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* and *data plane nodes*. The *control plane nodes* have a collection of processes which help monitor and maintain the desired state of the deployment. The *data plane nodes* contain processes which 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* [19]. 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 [17].

Figure 2.3: Overview of Kubernetes architecture



[1] https://kubernetes.io/

2.2.1.1 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 which 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 which searches for unscheduled pods using the API server. It then assigns the pods to a dataplane node based on several predicates and priorities such as resource requirements and node affinity respectively.

2.2.1.2 Data Plane

The *container runtime* is a process which 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 [2], containerd [3], and Docker [4].

[2] https://cri-o.io/
[3] https://containerd.io/
[4] https://www.docker.com/

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, perform 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.

2.3 Auto-scaling Overview

Apart from intelligently scheduling pods to data plane nodes, Kubernetes has the provisions to dynamically respond to changes in resource requirements [20]. This process of scaling nodes, pods, or other resources depending on requirements in an automated manner is known as *auto-scaling*. Kubernetes supports 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 Kubernetes deployments on external platforms such as Amazon [5] or Google [6]. The autoscaler works by looping through two tasks. The first watches for unscheduled pods, the second checks if the current deployed pods (pods which are running on the data plane) 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.

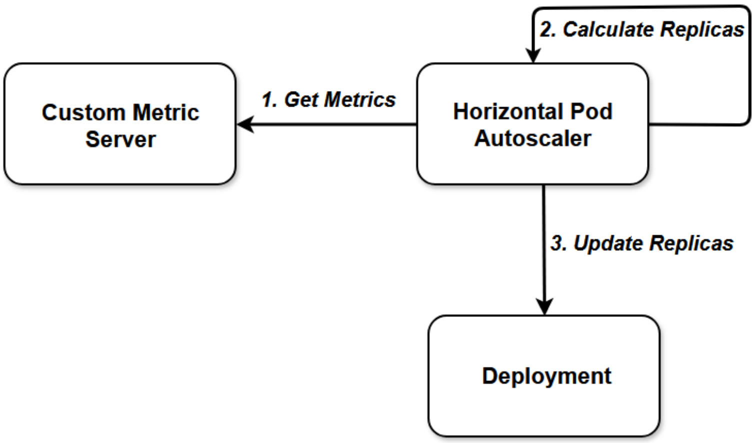
Horizontal pod auto-scaling is the most commonly used auto-scaling strategy [21]. It modifies the number of pods assigned to a task, based on the resources being requested. Kubernetes implements this using a periodic control loop which runs every 15 seconds by default. The control manager compares the actual resource utilization with the target utilization defined by the deployment script, and scales the number of pods accordingly.

[5] https://docs.aws.amazon.com/eks/

[6] https://cloud.google.com/kubernetes-engine/

2.3.1 Autoscaling Using Custom Metrics

Figure 2.4: Custom autoscaler architecture overview.



The default horizontal pod autoscaler uses pod CPU and memory utilization when making its scaling decisions. However, these metrics may be too rigid when it comes to scaling edge architecture resources [22]. The strict SLA constraints in place, along with the lower amount of resources present in the edge layer as compared to the cloud layer, make it imperative for custom metrics to be employed to autoscale resources as efficiently as possible.

Figure 2.4 depicts the general architecture of the custom autoscaler. Typically, the autoscaler queries metrics from the default metrics registry, which acts as a central store for all metrics that are exposed to the developer. Three interfaces to this registry are exposed:

- **Resource metric API**: This is used to access predefined metrics such as CPU and memory resources of both pods as well as nodes.
- **Custom metric API**: This contains user-defined custom metrics associated with all Kubernetes objects.
- **External metrics API**: This contains metrics of objects which are not associated with Kubernetes.

For custom metric auto-scaling, the autoscaler must be configured in a way where the metrics can be fetched from the custom metric API. This is done by configuring the custom metric server, several frameworks to simplify this process such as the Kubernetes Instrumentation SIG [7] exist which simplify the server building process.

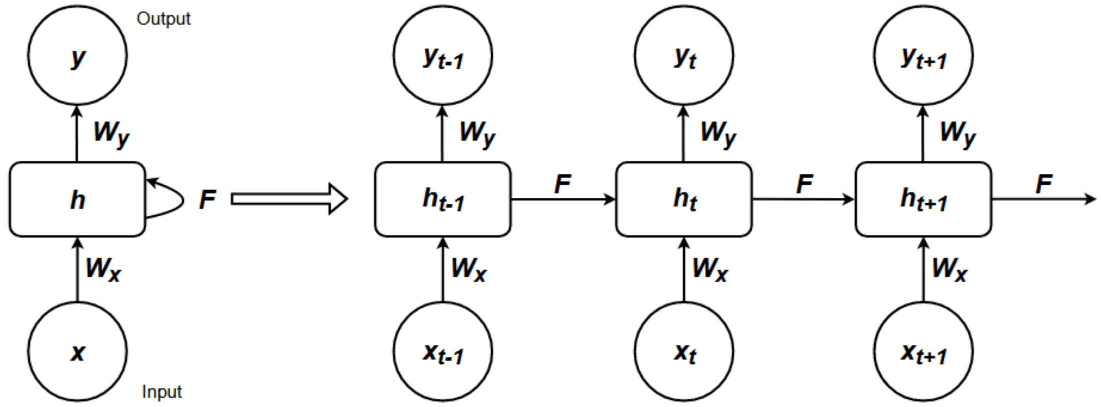
2.4 Time-Series Analysis

Several machine learning algorithms exist, however not all of them are suitable for time-series analysis [23]. One of the most popular traditional models is the Convolutional Neural Network (CNN), which is a form of deep learning that attempts to learn data features through the process of filters being applied to the input at each convolutional layer, and the output being passed to the next layer. However, while CNNs are excellent when working with 3-dimensional data such as images, it does not perform well when dealing with sequential inputs with interdependent data [24]. Due to this, another machine learning subset known as Recurrent Neural Network (RNN) was conceptualized.

2.4.1 Architecture Overview

Recurrent Neural Networks store information about the past, and its future decisions are based on this information. Thus, while RNNs have a similar training process as CNNs, they remember features learned from prior inputs. RNNs are recurrent since they perform the same computation for each data element in the sequence, with each output being dependent on the previous computation. Figure 2.5 depicts this high-level architecture of the RNN.

FIGURE 2.5: Recurrent Neural Network architecture



[7] https://github.com/kubernetes/community/tree/master/sig-instrumentation

For the current RNN state, the formula can be written as follows:

$$h_t = \mathcal{F}(h_{t-1}, x_t) \tag{2.3}$$

Where h_t is the current state, h_{t-1} is the previous state, and x_t is the current input value. In the simplest form, the function $\mathcal{F}$ is the activation function. Thus using the recurrent neuron weight W_h and input weight W_x, the equation can be re-written as:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t) \tag{2.4}$$

Using this current state value, the output can be calculated using the equation:

$$y_t = W_y h_t \tag{2.5}$$

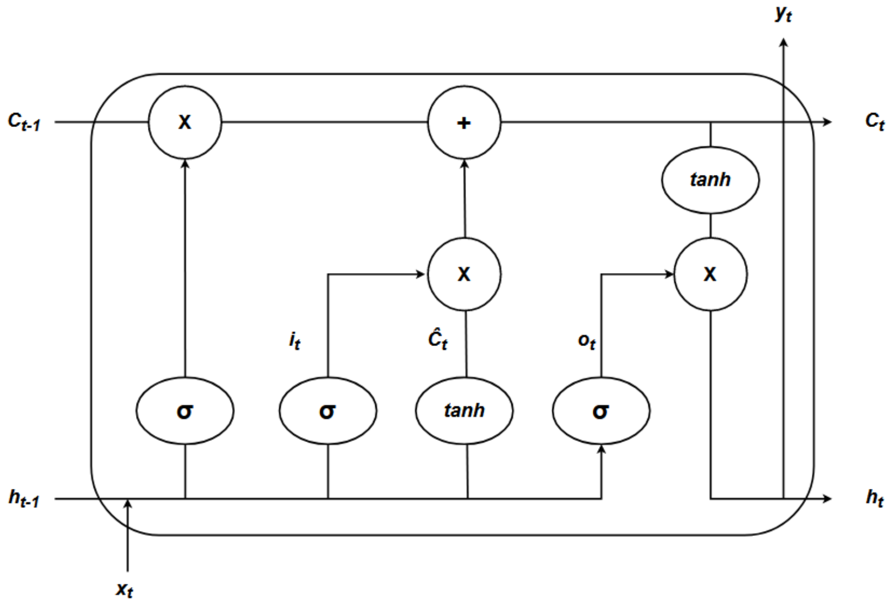
Where W_y is the output weight.

RNN also employs a back-propagation algorithm during training. However, the parameters are used by all the network states $h_i; i \in \{1, 2, ..., n\}$. However, RNN training encounters two big issues, namely the vanishing and exploding gradients, which hinder the accuracy of predictions. If the training sequence is too long, the RNN has a difficult time carrying previous information from one training iteration to the next, due to the exponential nature of the weight updates during back-propagation which causes its values to become extremely small or large. To address this issue, an improved version of RNNs was created, known as LSTM.

Long Short-Term Memory or LSTM is a deep learning model which is enhanced from the traditional RNN architecture [25]. In an RNN function, the hidden state activation is influenced by nearby activation functions. This is known as the "short-term" memory. Meanwhile the overall network weights are influenced by the computations which occur over the entire long data sequence.

LSTMs are designed in a way which avoids the long-term dependency problem by integrating a cell state into the architecture. The cell state acts as a conveyor belt of information, allowing information to be passed along the different data sequences easily.

Figure 2.6: LSTM cell architecture



Information is added to or removed from the cell state using structures known as gates. These gates are created from a sigmoid layer, and a point-wise multiplication operation. The LSTM architecture has three gates for performing cell state operations. These are the *Forget gate*, *Input gate*, and *Output gate*. Figure 2.6 shows the structure of these cell state operations.

The *forget gate* determines the amount of past information the LSTM should retain. This decision is made through the use of a sigmoid function, and the gate output f_t is given through the equation:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.6}$$

Where W is the gate neuron weight, and b_x is the gate bias.

The *input gate* determines how information is added to the cell state. This is done through the following steps:

- Determine the values i_t to be added:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.7}$$

- Create vector $\hat{C}_t$ of all the possible values that may be added:

$$\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.8}$$

- Use the computed value and vector, along with the previous cell state C_{t-1} to obtain the useful information which can be added to current cell state C_t:

$$C_t = f_t \times C_{t-1} + i_t \times \hat{C}_t \tag{2.9}$$

Finally, the *output gate* determines which portion of the current cell state is included in the output. This is determined using three steps:

- Create vector o_t which scales the cell state value between -1 and +1:

$$\hat{C}_t = \tanh(C_t) \tag{2.10}$$

- Compute filter value o_t which regulates which values from o_t are included in the output:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.11}$$

- Multiply the vector and filter to get the output h_t which also acts as the hidden state of the next cell:

$$h_t = o_t \times \hat{C}_t \tag{2.12}$$

2.4.2 Forecast Models

There are two major methods of using a time-series model to determine the predicted outcomes, namely a one-step forecast model, and a multi-step forecast model.

In a *One Step Forecast*, the time-series model first extracts short-term characteristics and correlations amongst the variables. It then feeds these into the various layers of its architecture during the training process, and finally generates a point forecast. In theory, such an approach would be more efficient, however it is heavily dependant on the correct training specifications and hyper-parameter selections, and thus is difficult to configure [26].

In a *Multi Step Forecast*, the time-series model does the same training steps as the above approach, but then it predicts several data points of output in one single shot. Such an approach may take less time to make a prediction, and has less chances of inaccurate predictions due to it being more robust to misspecifications in the model.

Chapter 3

Related Work

In this chapter, we discuss some of the common implementation strategies for edge computing paradigms in Section 3.1 before discussing the inherent challenges and limitations in Section 3.2. Then in Section 3.3 we review the resource scheduling solutions which attempted to resolve these challenges. Building on this foundation, Sections 3.4 to 3.6 comprise of a detailed literature review of the state-of-the-art auto-scaling algorithms used to mitigate these challenges, and a comparison of their performances and drawbacks.

3.1 Edge Computing Implementation

Yu *et al.* [27] states when implementing edge computing architectures, researchers typically focus on two models, namely the hierarchical model, and the software-defined model.

3.1.1 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 [28] gives an overview of such a hierarchical model. The bottom tier is usually comprised of a geo-distributed servers which receive workload from mobile devices via wireless links. These servers are then connected to a higher tier level of servers which comprise of the 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.* [29] proposed an edge cloud hierarchy which 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 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.* [30] demonstrated a similar approach to a hierarchical model, which integrated Mobile Edge Computers (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.

3.1.2 Software-defined Model

The number of end users and devices which connect to the edge architecture typically numbers in the millions [27]. Thus, management of an application deployed on this architecture can prove to be significantly challenging. To address these complexities, Software Defined Networks (SDN) were proposed.

According to Wang *et al.* [31], 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 [32]. 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 [33] presented an MEC model which 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.* [34] proposed an SDN model which 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.* [35] 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.

3.2 Edge Computing Issues and Challenges

3.2.1 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 multi-regional zones, or in the best case, globally. This is done so as to perform in-depth analysis in diverse fields such as health-care, robotics, and business decision making. Traditionally, they also dealt with non-real-time data for decision-making [36]. On the other hand, edge computing usually handles smaller scale data, locally clustered and isolated in separate zones, and highly real-time in nature [37]. 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 centres 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. As mentioned in Section 1.1, micro-service container orchestration technologies are leveraged to achieve these aims. Kristiani *et al.* [38] 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.* [39]. 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 resources allocation, violating edge architecture paradigms.

3.2.2 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 [40]. Even though the pod start up 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 initialising 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 [41], lazy fetching of container images [42], and container queues [43]. 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*.

3.2.3 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 [44].

- Degradation of application performance due to peak levels of workload, leading to user dissatisfaction [45].
- Incorrect resources being allocated to the application, leading to either a degradation of availability, or large cost of application deployment [46].

Several strategies have been proposed to counteract these challenges. Linlin *et al.* [47] 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.* [48] 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.* [45] 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.

3.3 Scheduling Strategies

To counteract the limitations discussed in Section 3.2.1, several custom scheduling and resource management algorithms have been proposed for edge architectures.

Skarlat *et al.* [49] demonstrated an algorithm for resource scheduling where the usage of resources was formalized as an optimization problem. Thus, the authors attempted to minimize the network delay when requesting computational resources. Based on this work, Aazam and Huh [50] proposed another solution for resource management which attempted to estimate the resources per service required, based on the user's previous behaviour as well as type of service being requested. By following such an approach, the resource wastage was actively reduced in edge nodes. Another resource allocation solution provided by Ni *et al.* [51] was based on priced timed Petri nets. The resources in the edge nodes are divided into several groups. The users can then select resources as per their requirements in an autonomous manner according to the price and time-cost of the operation.

Based on these initial proposals, Nguyen *et al.* [52] revealed ElasticFog, a resource provisioning algorithm which operates on top of the Kubernetes architecture. The algorithm provides real-time elastic scheduling for the resources on the edge nodes by monitoring the traffic distribution on the network, thereby achieving a significantly higher throughput and network efficiency in comparison to the default Kubernetes scheduler solution.

Wojciechowski *et al.* [53] proposed a similar extension to the Kubernetes scheduler which deployed traffic-aware provisioning of resources. The extension worked alongside the Istio Service Mesh [1] to collect dynamic network metrics for the scheduling of resources. The algorithm was shown to have highly efficient uses for edge deployments such as the 5G network.

While these proposals improved the scheduling and resource provisioning of edge deployments, they did not address key limitations addressed above such as the need for dynamic resource scaling so as to face the challenges of real-time data processing and SLA compliance. Due to these issues, several auto-scaling algorithms extending the design principles of these resource scheduling algorithms were proposed to address them.

3.4 Reactive Autoscaling Strategies

Nunes *et al.* [61] 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.* [62] stated

[1] https://istio.io/latest/about/service-mesh/

TABLE 3.1: Summary of reactive auto-scaling solutions

Features	**Reactive algorithms**							
	[39]	[54]	[55]	[56]	[57]	[58]	[59]	[60]
Simple deployment	✓	✗	✓	✓	✓	✓	✓	✓
Simple parameter tuning	✓	✗	✓	✗	✓	✓	✓	✗
Custom metrics	✗	✗	✗	✗	✗	✓	✓	✗
Light-weight deployment	✓	✓	✓	✓	✗	✗	✗	✓
Edge architecture compliant	✓	✓	✓	✓	✓	✓	✓	✓
SLA-compliant	✗	✗	✗	✗	✗	✗	✗	✗
Minimizes deployment cost	✗	✗	✗	✗	✗	✗	✗	✗

that this was due to the simplicity and user-friendliness in developing them. Table 3.1 summarizes the reactive auto-scaling algorithms discussed below:

Kampars and Pinka [54] 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, resulting in several improvements in performance. This logic however is complex, while the challenging metric selection and integration procedure make it unsuitable for deployment on an edge architecture.

Zhang *et al.* [55] 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 which 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 to balance system stability with a decent elasticity required careful tuning of parameters such as the cooldown periods of scaling. However the lack of addressal of the cold start problem discussed above results in a delay in scaling resources, violating SLA-compliance.

Srirama *et al.* [56] investigated an container-aware auto-scaling solution which deploys applications to containers which 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. However the complex and user-intensive parameter tuning make it difficult to adapt to generalized use-cases.

Hoenisch *et al.* [57] implemented a four-fold auto-scaling strategy for containerised 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%. The drawback is the heavy performance and resource overhead when running in edge deployments, leading to a degredation in application performance.

Santos *et al.* [58] 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 which can be customized by the end-user. While the authors explained that the experimental results displayed a performance comparable to other reactive solutions, further research demonstrated that the algorithm is unable to scale well on highly dynamic workloads.

Sheganaku *et al.* [59] devised an container-based auto-scaling solution which allocates resources in a four-fold manner similar to Hoenisch *et al.* [57]. 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. However, this came at a cost as the solution is too time consuming and computationally expensive for edge deployments.

Taherizadeh and Stankovski [60] proposed a multi-level auto-scaling solution using a rule-based approach. The algorithm uses dynamically changing thresholds based on both the container infrastructure as well as application, resulting in improved performance as compared to other reactive approaches, however such a multi-level solution is difficult to tune and optimize, making it unsuitable for dynamic and custom workloads.

Phan *et al.* [39] 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. 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 modelling 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. However the algorithm is not SLA compliant due to the delay in scaling resources in a reactive manner.

Table 3.2: Summary of proactive auto-scaling solutions

Features	Proactive algorithms					
	[63]	[64]	[65]	[66]	[67]	[68]
Simple deployment	✗	✓	✓	✓	✓	✓
Simple parameter tuning	✗	✗	✗	✓	✗	✗
Custom metrics	✓	✓	✗	✗	✗	✗
Light-weight deployment	✗	✗	✗	✓	✗	✗
Edge architecture compliant	✓	✗	✗	✗	✗	✗
SLA-compliant	✗	✓	✓	✓	✗	✗
Minimizes deployment cost	✗	✗	✓	✗	✗	✗

3.5 Proactive Autoscaling Strategies

Lorido *et al.* [69] 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 populary 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 non-linear data [62]. The development of a generalised 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 [70], though modern algorithms ensure that they avoid this pitfall [71]. Despite these challenges, their application in scaling of resources with semi-predictable data series remain valuable. Table 3.2 summarizes the proactive auto-scaling algorithms discussed below:

Ju *et al.* [63] 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. 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, the results concluded that this algorithm significantly outperformed both the default Kubernetes autoscaler, as well as existing reactive autoscaling solutions. 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.* [64] 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%. This showed a high prediction accuracy, however the cost of training this model was prohibitively high, making it unsuitable for edge deployments.

Imdoukh *et al.* [65] 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. However, this algorithm also suffers from the problems related to a lack of initial training data encountered by Ju *et al.* [63]

Messias *et al.* [66] created a proactive autoscaler using genetic algorithms (GA). 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. The initial state of the genetic algorithm is however set randomly [72], this causes the initial forecasting to contain a significant amount of errors, and the algorithm can take an arbitrarily large amount of time to converge to an acceptable forecast result.

Abdulla *et al.* [67] devised an auto-scaling solution which 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. The XGBoost algorithm used to forecast these burts is quite rigid however, and thus is incapable of capturing several other workload patterns.

Alidoost *et al.* [68] 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. The SVM however struggles to identify boundaries for complex data, making this algorithm unsuitable for non-linear workload patterns.

3.6 Hybrid Autoscaling Strategies

All the approaches mentioned in Sections 3.4 and 3.5 have their benefits and drawbacks. Thus, hybrid solutions which merge multiple auto-scaling methods were proposed [78]. 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. Despite this, exploring these solutions provides a solid template for the approach used in this paper, Table 3.3 shows an overview of the proposals discussed below:

Table 3.3: Summary of hybrid auto-scaling solutions

Features	**Hybrid algorithms**					**Proposed solution**
	[73]	[74]	[75]	[76]	[77]	
Simple deployment	✓	✓	✓	✓	✓	✓
Simple parameter tuning	✓	✓	✓	✗	✗	✓
Custom metrics	✓	✗	✗	✓	✓	✓
Light-weight deployment	✓	✗	✓	✗	✗	✓
Edge architecture compliant	✗	✗	✗	✗	✗	✓
SLA-compliant	✗	✗	✓	✓	✓	✓
Minimizes deployment cost	✗	✗	✗	✗	✓	✓

In 2007, one of the first hybrid algorithms for a distributed deployment was proposed by Jing *et al.* [73]. 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 [74] proposed a resource provisioning algorithm for multi-cluster set ups using a hybrid autoscaler. The autoscaler comprised of a combination of fixed fuzzy rule-based logic and a self adaptive algorithm which dynamically tuned the scaling factor. The authors tested this algorithm on a simulation to demonstrate performance benefits compared to existing approaches. This however was a limited experiment, not comprising of tests on a production environment.

A hybrid approach for cloud computing architectures was proposed by Rampérez *et al.* [75]. The algorithm which was 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. FThe linear regression forecaster was however too simplistic to predict complex time-series, leading to erroneous results.

Biswas *et al.* [76] 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. Such an SVM-based model is expensive to train however, making it infeasible to deploy on edge deployments due to the resource and latency constraints discussed above.

Finally, another cloud computing based autoscaler with SLA-constraints was proposed by Singh *et al.* [77]. 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) [79]. 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. However, the TASM forecaster was too complex and resource intensive to be deployed on a scarce resource padadigm such as the edge layer. This resource intensiveness increased the training times drastically too, making it infeasible for conforming to SLA constraints on an edge architecture paradigm.

The above algorithm by Singh *et al.* [77] provided the best approach to be used when creating a hybrid autoscaler. Therefore, while implementing the autoscaler presented in this thesis, the generalized architecture of the RHAS deployment was further extended to streamline both the reactive and proactive autoscalers, while choosing a more efficient and cost-effective forecasting model so as to make it SLA-compliant on edge architectures, and eliminating the costly and time-consuming hyper-parameter tuning process.

Chapter 4

Problem Formulation and Autoscaler Design

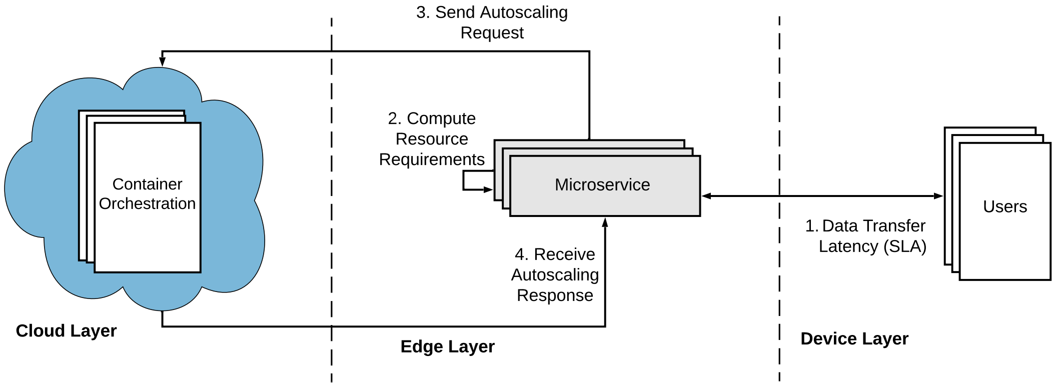
Based on the overview of and challenges faced by edge computing paradigms in conforming to SLA constraints, the stated research questions, and the related works discussed above, a hybrid auto-scaling algorithm to answer these questions was proposed during the course of this research project.

In this chapter, Section 4.1 will discuss and formulate the problem of auto-scaling resources on the edge architecture in an SLA-compliant manner. Section 4.2 will then detail the high-level overview of the proposed hybrid autoscaler, the objectives of the various algorithm subsystems, the challenges it faces, and an analysis of the algorithm's space and time complexity.

4.1 Problem Overview

Edge architectures are split into three layers [80]. 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 4.1: Autoscaling problem overview


3. Send Autoscaling Request
2. Compute Resource Requirements
Container Orchestration
Microservice
Users
1. Data Transfer Latency (SLA)
4. Receive Autoscaling Response
Cloud Layer
Edge Layer
Device Layer

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. However, the primary drawback is the distance between the user and the cloud layer which results in significant latency, making it unsuitable for serving real-time user requests. Thus 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 micro-service which receives and serves user data. These worker nodes allocate resources to the micro-service deployments dynamically according to user requirements through the process known as auto-scaling.

Figure 4.1 shows the auto-scaling process. The users in the device layer send requests and receive responses to the micro-service deployment in the edge layer. The time taken to receive this response is considered to be the SLA latency metric negotiated between the edge deployment provider and the customer. Using the number of requests being received by the user layer, the edge layer micro-service autoscaler computes the total resource requirements to serve the customer. Based on its findings, the edge layer requests the container orchestration in the cloud layer to either downscale or upscale its resources. The container orchestration then computes the required resource allocation as well as the nodes on which to allocate them and sends its response back to the edge layer.

Table 4.1: Definition of Symbols

Symbol	Definition
$\mathcal{S}_c(t)$	SLA constraint metric value at time t
Δ	Max SLA metric threshold agreed by the cloud provider & customer
$\mathcal{V}$	Number of SLA violations for the chosen metric
$\mathcal{C}_t$	"Cold start" time taken to scale resource replicas
$req_{RT}(t)$	Round trip latency of user request to edge layer
$\mathcal{K}_t$	Constant latency present between edge and device layers
$\mathcal{U}(t)$	Latency of deployment with unitary resource replica
$\mathcal{D}$	Total resources in micro-service deployment
p_i	Resource pod i of deployment $\mathcal{D}$ where $i = \{1, 2, ..N\}$
α	Cost per unit resource for cloud provider
$\mathcal{L}$	Maximum allowed deployment cost assigned by the user
$\mathcal{R}(t)$	Time taken to scale replica resources

4.1.1 Problem Formulation

Cloud deployments provide several Quality of Service (QoS) metrics when considering SLA negotiations [81]. These can be broadly classified into the following four categories:

- **Performance**: These are metrics such as the *response time* which computes the total time taken for a user request to be processed, or the *throughput*, which shows the cloud scalability.
- **Availability**: These include the *abandon rate* which shows the ratio of dropped service requests to the total user requests, and the *use rate* which calculates the amount of time a cloud service was used. Furthermore, certain performance metrics such as *response time* exceeding their thresholds can be considered availability metrics as well.
- **Reliability**: This includes metrics such as the *mean failure time* which calculates the predicted time between service failures, and the *mean recovery time* which is the average time it takes for the service to recover from said failure.
- **Cost**: This includes *financial* costs of deploying or using a cloud service, as well as *energy* costs which compute the carbon footprint of running a cloud service.

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. Reliability and cost metrics such as *energy* do not correlate well to auto-scaling, while availability metrics such as *abandon rate* are too vague in their reading, showing a binary value of the request either being dropped, or processed. 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.

$$\mathcal{S}_c(t) > \Delta \tag{4.1}$$

The threshold Δ shown in Equation 4.1 varies from application to application. Hussain *et al.* [82] provided a framework wherein multiple SLA thresholds were configured by the cloud provider and customer, so as to apply them for a variety of use cases, and to ensure differing responses based on the severity of the violation. For a response time latency based metric, these thresholds can broadly be classified into three categories.

- **Flexible**: This is typically the highest allowed violation threshold for the application. Flexible SLA metrics are used to gauge the availability of the deployment. Most IoT applications employ this threshold.
- **Moderate**: This threshold is a trade-off between flexible and stricter SLA thresholds. This threshold is used by applications to ensure a real-time capability such as traffic light scheduling in railways.
- **Strict**: This is the lowest allowed violation threshold in the application. This threshold is significantly challenging to maintain and is used by extremely time-critical applications such as remote-controlled tools for medical surgeries.

The auto-scaling will thus use a resource metric to scale its resources up or down. The autoscaler will check to see if the micro-service metric exceeds the threshold for a certain time period, and if so, autoscale resources accordingly. A problem arises in the time it takes to scale these resources, however. This time to increase the number of resource replicas $\mathcal{R}$, which we define as the cold start time $\mathcal{C}(t)$ which can be defined as:

$$\mathcal{C}(t) = \mathcal{R}_{download}(t) + \mathcal{R}_{deploy}(t) + \mathcal{R}_{register}(t) \tag{4.2}$$

Here, the cold start time $\mathcal{C}(t)$ is defined as the summation of the time taken for the replica $\mathcal{R}$ to be downloaded, deployed on the data plane, and registered with the control plane for scheduling and accepting user requests. The replica image is typically downloaded from a public repository. This download time is usually a one-time delay due to optimizations done on modern container orchestration software, and can be ignored for SLA latency calculations. Using this information, we can reduce this equation to the following:

$$\mathcal{C}(t) \approx \mathcal{R}_{deploy}(t) + \mathcal{R}_{register}(t) \tag{4.3}$$

Here, the time to deploy and register the replica to the container orchestration cloud layer cannot be avoided. Furthermore, it can be shown that the number of SLA violations $\mathcal{V} \propto \mathcal{C}(t)$ due to the correlation between cold-start delay and the lack of available resources [83].

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

$$\mathcal{S}_c(t) = \mathcal{C}(t) + req_{RT}(t) \tag{4.4}$$

This round-trip time $req_{RT}(t)$ is the combined sum of the inherent delay present in the network layer $latency_{N/W}(t)$, and the time taken for the edge application to process the request $processing_{app}(t)$.

$$req_{RT}(t) = 2 \times latency_{N/W}(t) + processing_{app}(t) \tag{4.5}$$

The network delay can be reduced by investing in higher network bandwidths, but such improvements have a maximum physical limit, which would not solve the cold start issue.

Here we consider this latency to be a constant $\mathcal{K}(t)$. Furthermore, $processing_{app}(t)$ is inversely proportional to the available resources to the application $\mathcal{D}$, since increasing the number of resource replicas helps spread out the user request workload thus reducing the chances of a bottleneck. Using this information, $req_{RT}(t)$ can be approximated as:

$$req_{RT}(t) \approx \mathcal{K}(t) + \frac{\mathcal{U}(t)}{\mathcal{D}} \tag{4.6}$$

Where $\mathcal{U}(t)$ is the maximum latency of a unitary resource deployment. For horizontal pod auto-scaling, the resources here are the number of pods in deployment resources $\mathcal{D}$ such that $\mathcal{D} = \sum_i p_i$, where i represents the current number of active pods. These pods are the smallest unit of resource for auto-scaling purposes that process the user requests. The final SLA equation can be re-written as follows:

$$\mathcal{S}_c(t) = \mathcal{C}(t) + \frac{\mathcal{U}(t)}{\sum_i p_i} + \mathcal{K}(t) \tag{4.7}$$

Thus, the primary aim of the autoscaler is to significantly reduce or even eliminate the cold start, while also increasing the number of resources assigned to the deployment to minimize the application latency. By doing so, the autoscaler aims to reduce the SLA latency below the agreed threshold Δ thus minimizing the number of SLA violations $\mathcal{V}$.

From the above equation, it is clear that $\lim_{\sum_i p_i \to \infty} \dfrac{\mathcal{U}(t)}{\sum_i p_i} = 0$.

Thus, the equation incentivizes ignoring intelligently auto-scaling altogether and simply allocate the maximum number of pods to the deployment. However, there are drawbacks to doing so.

Most cloud providers such as Amazon Web Services and Google Cloud Platform allocate a cost for each resource assignment to the deployment. If the maximum number of pods that can be deployed is N, the cost is calculated as follows.

$$cost = \alpha \times \sum_i p_i \quad ; i \leq N \tag{4.8}$$

Where α is the unitary resource cost which may vary depending on the cloud provider. Thus, simply scaling all resources to the maximum amount may result in substantial and infeasibly high deployment costs. Based on this additional information, the auto-scaling SLA Equation 4.7 and the cost Equation 4.8 can be merged to form an optimization problem $\mathcal{P}$.

$$\mathcal{P} = x \times \mathcal{S}_c(t) + y \times cost \tag{4.9}$$

The objective of the autoscaler is to assign resources in a way that minimizes both the latency, as well as the cost, thus minimizing $\mathcal{P}$. The parameters x and y dictate how important cost and latency are relative to each other when considering auto-scaling. For this research, we will configure these values as $x = y = 0.5$, implying both are equally important. Furthermore, the customer will have a maximum "budget" $\mathcal{L}$ which the deployment must not exceed. Maximizing the value of the resources in $\mathcal{D}$ while limiting the cost below the threshold $\mathcal{L}$ to reduce the SLA constraint metric $\mathcal{S}_c$ below the agreed threshold is akin to the famous Knapsack Problem [84]. This problem is proven to be NP-Hard, and as such no known algorithm can determine the best value in polynomial time as demonstrated by Kellerer *et al.* [85]. However, an approximation close to this best value can be computed instead, and this can be done in polynomial time. Due to this, most autoscalers rely on a reactive rule-based or proactive machine-learning technique to compute such close approximations.

A problem however arises in the amount of resources it takes to train a proactive model. Not only does the training process consume a significant amount of CPU and memory resources, but it also requires a large time-series dataset. This dataset is necessary to generate a sufficient number of training windows. Without enough training windows, the model will produce erroneous results. While hybrid models help to mitigate the initial errors via the reactive auto-scaling component, the questions regarding the proactive model's resource usage remain an open issue [86].

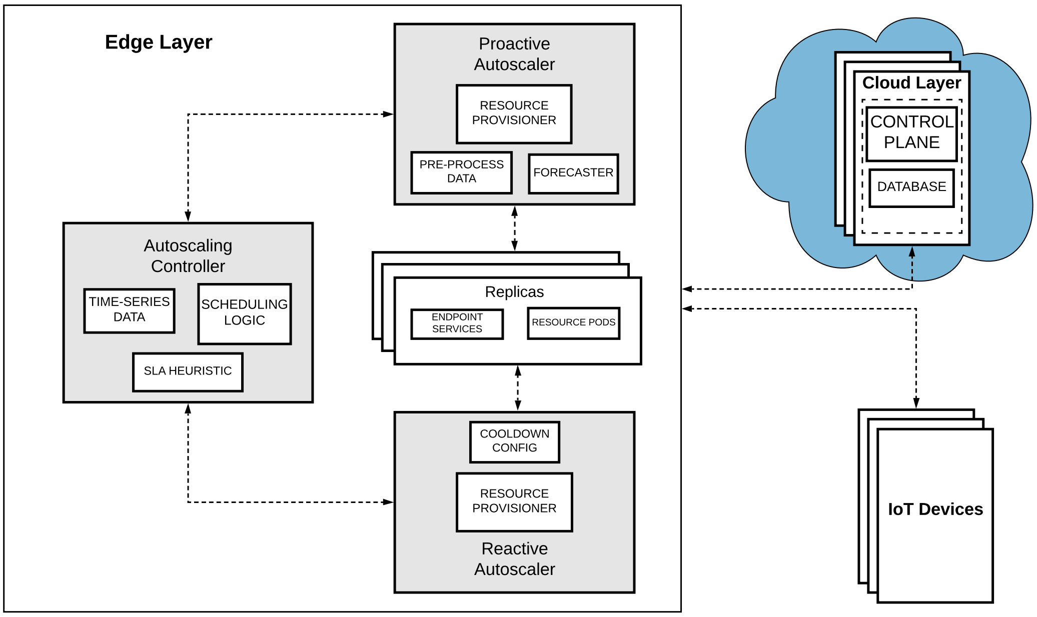
Another issue in a purely proactive autoscaler is the amount of time it takes to both train the model, as well as generate the predictions. Torres *et al.* [87] have demonstrated that time-series forecasters such as LSTM and ARIMA are known to typically have incredibly complex deep neural networks with thousands of neurons in several layers.

They demonstrated that even when running such a deep neural network on a cloud architecture, with the aid of 6 cores of CPU and 16 GB of memory resources, and additional parallelization through the use of modern graphical processing units (GPU), this can result in the training process taking upwards of an hour. A large training time makes it incredibly difficult to actively tune the hyper-parameters of the model based on the perceived accuracy, as adapting these values too frequently would result in several hours of training per day, during which the model is unable to predict new values. Once again, a hybrid architecture is typically used to mitigate this issue, as when the proactive model is training, the reactive autoscaler takes over. However, this approach is not feasible for an SLA-constrained architecture, as the reactive component is not SLA-compliant due to its inability to mitigate the cold start problem.

In most proactive autoscalers, the forecaster attempts to accurately model the time-series curve to allocate resources effectively. Even in the hybrid algorithms that have been proposed in Section 3.6, the proactive modules of the autoscalers are generally unmodified proactive forecasters bundled together with a reactive component. This strategy of attempting to perfectly forecast the curve is what takes such large amounts of resources.

Hence, the autoscaler proposed in this thesis will not attempt to predict the exact resource workload. Instead, the time-series graph is heavily simplified using a noise filtering method and then inputted to the machine learning model. Furthermore, the time-series forecaster only attempts to predict the exact timestamp when resource requirements start to increase. Every other requirement, such as the stable resource utilization, as well as the drop-off in non-peak time periods can be handled by the reactive autoscaler, thus heavily simplifying the forecaster architecture. This drastically reduces the forecaster training time to a few minutes. The simplified forecaster has the additional benefit of not requiring incredibly lengthy amounts of time-series data to be stored for it to make accurate predictions, thus this data can also be kept in the edge layer. This makes the autoscaler extremely lightweight, responsive, and most importantly SLA-compliant, thus making it capable of being deployed in an edge environment.

Figure 4.2: Proposed hybrid architecture overview



4.2 Proposed Hybrid Autoscaler

An overview of the hybrid autoscaler architecture is shown in Figure 4.2. The overall architecture is formulated using a hierarchical model as described in Section 3.1. 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.* [55] demonstrated, the micro-service system stability is directly related to the careful selection of cool-down parameters. Thus, these must be available to the user in a configuration setting.

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. A detailed implementation of the forecaster logic itself will be discussed in Section 5.3.3.

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 also hosts the time-series metric data and has a feedback loop with the proactive autoscaler. If it detects any SLA violations were caused after auto-scaling during a configured time window, it automatically adjusts the hyper-parameters of the proactive forecaster in an attempt to predict from the time-series data more accurately during the next training iteration. Correspondingly, a lack of SLA violations during a specific time window period reverts the autoscaler parameters back to the original values, in an attempt to streamline the training process further. Such a heuristic method allows for the freeing up of the complex hyper-parameter tuning process seen in most proactive models from the autoscaler deployment process. This is a key part of the architecture which is essential in answering the research questions outlined in the thesis.

4.2.1 Autoscaler Subsystems

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

$$replicas_{desired} = \lceil replicas_{current} \times \frac{metric_{current}}{metric_{desired}} \rceil \tag{4.10}$$

For example, for a given deployment with a current replica count as 1, if the desired metric value is 50 resource units, and the current value is 100, then the number of desired replicas will be $\lceil 1 \times \frac{100}{50} \rceil = 2$. There are three other important parameters that are key to controlling the process of horizontal pod scaling, namely "tolerance", "scale up cooldown", and "scaledown cooldown".

The tolerance is a constant which informs the autoscaler when to skip calculating new replicas. The tolerance ratio can be calculated as:

$$tolerance = \left| \frac{metric_{desired} - metric_{current}}{metric_{desired}} \right| \tag{4.11}$$

For example, if the current metric is 60, and the desired metric is 50, the tolerance is calculated as $tolerance = \left|\frac{60-50}{60}\right| = 0.167$. By default, if the tolerance value is below 0.1, autoscaling is skipped for that control loop, however, this can be configured by the user.

The scale-up and scale-down cooldowns control how quickly auto-scaling occurs. The default approach which is set by the container orchestration can be concisely stated as "Scale up as quickly as possible, while scaling down very gradually." Therefore, the default scale-up cooldown is set to 0 seconds, meaning that the moment the desired replica value increases, the autoscaling will be initiated. However, the default cooldown is set to 300 seconds, meaning that if the desired replica value is decreased, it must remain decreased for 300 seconds (or 20 control loops) before the resources are scaled down.

A cooldown value that is too low would cause a repetitive scaling up and scaling down of the resources, leading to significant stress on the system as well as wastage of resources. Meanwhile, a large value would render the autoscaler unable to assign resources quickly enough to ensure SLA latency compliance. Thus, for the proposed autoscaler, the default values are modified to ensure that a moderate cooldown value was chosen to ensure the best system stability and SLA compliance. This cooldown configuration would be applied to both the proactive and reactive auto-scaling subsystems to maintain consistency.

For auto-scaling proactively, a custom metric $metric_{forecast}$ is used, which defines the future CPU workload expected to be exerted on the micro-service application $\mathcal{T}$ seconds in the future, where $\mathcal{T}$ can be configured during the autoscaler deployment process. This $metric_{forecast}$ value will be sent to the auto-scaling controller by the proactive autoscaler.

4.2.1.1 Scheduler

The autoscaler controller consists of a scheduling logic module which handles when to switch between proactive and reactive auto-scaling. Algorithm 1 explains how the hybrid scheduling logic determines which auto-scaling subsystem to employ. The hybrid scheduler takes four inputs, namely the $replicas_{current}$, $metric_{current}$, $metric_{desired}$, and $metric_{forecast}$ variables discussed above. It outputs one value, the $replicas_{desired}$.

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, which determines the current resource requirements. By computing the future requirements $replicas_{forecast}$ using Equation 4.10, if it is found that this requirement is higher than the current resource requirement $replicas_{current}$, then the hybrid scheduler outputs the forecaster replica count as the desired replicas. Otherwise, the hybrid scheduler determines that the utilization is either stabilizing or about to decline. Due to this, it makes a decision that the reactive replica count $replicas_{reactive}$ is the desired number. The hybrid scheduler then sends this value to the container orchestration controller plane to autoscale the replicas accordingly using either the reactive or proactive sub-system's resource provisioning modules.

Algorithm 1 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 $replicas_{desired}$
```

4.2.1.2 Reactive Resource Provisioning

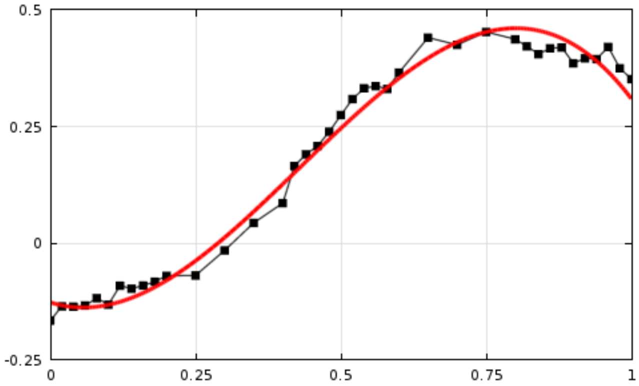
The reactive autoscaler subsystem is responsible for determining whether or not autoscaling should proceed based on the given configuration. The reactive algorithm's resource provisioning is built on top of the default horizontal pod autoscaler deployed by the Kubernetes container orchestration platform. The autoscaler is modified in such a way that it has its cooldown parameters set to a moderate value to ensure adaptability to SLA-constrained scenarios, while also maintaining system stability. The workflow is shown below in Algorithm 2. The algorithm takes the current metric value as an input, as well as the desired metric value, and outputs the decision to autoscale or not. It does this by computing the *tolerance* value as shown in Equation 4.11. If this tolerance is above the configured threshold, the autoscaler will modify the replicas, otherwise, it will ignore the current auto-scaling request.

Algorithm 2 Reactive resource provisioning

```
Input: $metric_{current}$, $metric_{desired}$
Output: $autoscale$
  $tolerance \leftarrow \left| \frac{metric_{desired} - metric_{current}}{metric_{desired}} \right|$
  if $tolerance > \gamma$ then
      $autoscale \leftarrow Yes$
  else
      $autoscale \leftarrow No$
  end if
  return $autoscale$
```

4.2.1.3 Data Pre-Processor

FIGURE 4.3: Pre-processing of data, source: [88]



To speed up the forecast process even further and reduce the resource and time requirements, the time-series data can be pre-processed to smoothen it. Doing so makes it easier for the deep learning model to extract patterns, and reduces the training and validation loss. Figure 4.3 shows a graph containing raw input (shown in black), and a smoothened data curve (shown in red). While the red curve contains all the requisite information of the data (such as the slope of the curve, maximum and minimum value, etc.), it removes the noise, reducing the overall loss, and reducing the length of the training window data sequence the LSTM requires to accurately predict future data.

4.2.1.4 Proactive Forecaster

The forecaster portion of the autoscaler is used to generate this $metric_{forecast}$ value. The autoscaler controller periodically scrapes the windowed data stored in the cloud database to keep a form of cached time-series workload. It combines this workload with the future workload $metric_{forecast}$. To do so, the controller requests the $metric_{forecast}$ value for a specified time $\mathcal{T}$ from the proactive forecaster, and the forecaster then sends this value back after pre-processing the scraped data and training the forecaster model.

Several time series forecaster algorithms exist, with the two prominent ones being the more modern deep learning algorithm LSTM, and the traditional deep learning algorithm ARIMA. Siami-Namini *et al.* [89] demonstrated that LSTM implementations outperformed ARIMA, reducing error rates by over 80%. Furthermore, they were able to demonstrate that the number of deep learning "epochs", or the total amount of training time required for LSTM did not need to be set to a high value. In fact, setting a significantly higher value than required was shown to degrade performance due to over-fitting. The authors posited that LSTM worked so well due to the "rolling updates" being performed on the model. The LSTM weights are only set once when the forecaster is deployed, after which they are always updated on every call of the training algorithm, meaning there is a continuous improvement to the prediction results.

Based on the investigations above, it was determined that LSTM time-series forecasters would be ideally suited for a proactive autoscaler designed for the hybrid autoscaler architecture. Algorithm 3 shows the implementation of such a forecaster. The autoscaler controller implements a control loop every $\mathcal{P}$ seconds, where it requests the latest time-series metric prediction from the forecaster. As input, the algorithm takes the variable $lookback$, which is the number of data points it should use as a window to train, the training iteration count $epochs$, and the $learning\_rate$, which determines the step size per each training iteration moving towards the minimum loss value. The forecaster then pre-processes this data to remove the noise as shown above in Section 4.2.1.3, and performs a training iteration using the configured hyper-parameters. It then computes the validation loss, and accepts this model as the most accurate one if it has a lower validation loss than the previous training iterations. Otherwise, the current model is rejected and the forecaster uses the older model instead. Finally, the model predicts the future metric and returns it to the autoscaler controller for determining the auto-scaling based on Algorithm 1.

Algorithm 3 Proactive forecaster

```
Input: $lookback \geq 0, 0 \leq epochs \leq 100, 0 \leq learning\_rate \leq 1$
Output: $metric_{forecast}$
  $lstm\_model \leftarrow lstm.initialize()$
  $time\_series \leftarrow get\_latest\_data()$
  $lstm\_input \leftarrow get\_input(time\_series\_data, lookback)$
  $lstm\_input \leftarrow preprocess\_data(lstm\_input)$
  $new\_model \leftarrow train(lstm\_input, epochs, learning\_rate)$
  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 $metric_{forecast}$
```

4.2.1.5 Proactive Resource Provisioning

The proactive subsystem's resource provisioning module works in a similar manner to the reactive autoscaler. If the autoscaler controller's scheduling logic determines that a proactive auto-scaling must take place, it requests the cloud layer to proceed with auto-scaling using the proactive subsystem's resource provisioning, which is shown in Algorithm 4. The algorithm takes the desired and forecast metric values as input and outputs the decision to autoscale or not. It does this by computing the *tolerance* value as shown in Equation 4.11. If this tolerance is above the configured threshold, the autoscaler will modify the replicas, otherwise, it will ignore the current auto-scaling request.

Algorithm 4 Proactive resource provisioning

```
Input: $metric_{forecast}, metric_{desired}$
Output: $autoscale$
  $tolerance \leftarrow \left| \frac{metric_{desired} - metric_{forecast}}{metric_{desired}} \right|$
  if $tolerance > \gamma$ then
      $autoscale \leftarrow Yes$
  else
      $autoscale \leftarrow No$
  end if
  return $autoscale$
```

4.2.1.6 SLA Heuristic Feedback

Finally, the last hybrid autoscaler module is the autoscaler controller's SLA-based heuristic feedback loop, which assists the proactive forecaster in increasing its prediction accuracy. The autoscaler constantly checks for SLA violations in the edge deployment using a control loop. Typically, the SLA checks are done for a sufficiently lengthy period of time 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 a number of causes, such as insufficient training data, or the LSTM hyper-parameters being too conservative. To temporarily boost learning, the controller then decreases the learning rate to increase the probability of the model escaping from the local minima to find the global one, increases the batch size to reduce under-fitting, and increases the number of epochs to reduce loss. All these parameters have a threshold, as increasing or decreasing certain parameters by a large amount may lead to issues such as over-fitting or infeasibly lengthy training times.

Algorithm 5 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
      $batch\_size \leftarrow MAX(batch\_size + \alpha, \mathcal{A})$
      $learning\_rate \leftarrow MIN(learning\_rate - \beta, \mathcal{B})$
      $epochs \leftarrow MIN(epochs - \lambda, \mathcal{L})$
  else
      $epochs \leftarrow initial\_epochs$
      $learning\_rate \leftarrow initial\_rate$
      $batch\_size \leftarrow initial\_batch$
  end if
  $hyperparameters_{modified} \leftarrow (learning\_rate, batch\_size, epochs)$
  return $hyperparameters_{modified}$
```

Finally, if the feedback control loop discovers that no SLA-violations occurred during the time-period, it concludes that the LSTM has sufficiently learned the primary characteristics of the time-series. As discussed by Siami-Namini *et al.* [89], the "rolling-updates" feature of LSTM allows the autoscaler to safely reset the hyper-parameters of the model, while preserving the learning and weights of the previous rounds of training. Algorithm 5 shows how the heuristic feedback is set up. The algorithm takes the three hyper-parameters of the LSTM, namely *learning_rate*, *batch_size*, and *epochs*, along with the $\mathcal{V}$ which stores the number of SLA violations that occurred during a specified time window. Using these parameters, the algorithm computes the new LSTM hyper-parameters and outputs them to the proactive forecaster to use in the next training cycle.

4.2.2 Computational Complexity Analysis

To analyze the space and time complexity of the auto-scaling architecture, some simplifying assumptions must be made. Firstly, the complexities will be for one control loop of the container orchestration platform. By default, controller orchestration platforms have control loops which are executed every 15 seconds. During each of these loops, auto-scaling is assessed and implemented. Thus we will ignore the complexities involved in the control loop itself, and focus only on the autoscaler.

Secondly, for the proactive forecaster, the complexities of the LSTM are simplified to the basic multiplicative steps involved in the training step. Other internal logic such as the σ function is assumed to have $\mathcal{O}(1)$ space and time complexity.

Finally, assuming the hybrid autoscaler $\mathcal{H}$ takes a time-series data of length $\mathcal{N}$, it stores this data in an array data structure. The rest of the inputs can be considered as single variables. The forecaster's LSTM weights are assumed to be a two-dimensional matrix of size $A \times B$. Thus the space complexity can be easily computed for this information. The variables each have a space complexity of $O(1)$. Meanwhile, the time-series array has a complexity of $\mathcal{O}(N)$, and the weights are of complexity $\mathcal{O}(N^2)$. Thus, the space complexity can be written as the sum of the two values and simplified.

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

$$\Rightarrow Complexity_{space}(\mathcal{H}) = \mathcal{O}(N^2) \tag{4.12}$$

The time complexity is a more complex calculation. The complexities of the reactive autoscaler, proactive autoscaler, and autoscaler controller need to be computed. The autoscaler controller only stores the time-series array and computes the new hyperparameters in Algorithm 5. Both of these operations can be completed in constant time, thus the time complexity for the controller is as follows:

$$Complexity_{time}(controller) = \mathcal{O}(1) \tag{4.13}$$

Similarly, the reactive autoscaler only computes the tolerance value, which is also a constant operation independent of data size, thus the time complexity can be written as:

$$Complexity_{time}(reactive) = \mathcal{O}(1) \tag{4.14}$$

For the proactive autoscaler, the forecaster internally computes matrix multiplications, such as of the various LSTM weights and vectors. For example, Equation 2.7 can be rewritten as follows.

$$i_t = \sigma(W_i(x) \cdot x_t + W_i(h) \cdot h_{t-1} + b_i) \tag{4.15}$$

In the above equation, we assume the dimension of the variables as follows:

- $W_i(x) \in \mathbb{R}^{m \times n}$
- $x_t \in \mathbb{R}^n$
- $W_i(h) \in \mathbb{R}^{m \times m}$
- $b_i \in \mathbb{R}^m$
- $h_{t-1} \in \mathbb{R}^m$

Using these values, the time complexity of each of the individual multiplicative and additive operations can be computed.

$$Complexity_{time}(W_i(x) \cdot x_t) = \mathcal{O}(mn)$$

$$Complexity_{time}(W_i(h) \cdot h_{t-1}) = \mathcal{O}(m^2)$$

$$Complexity_{time}(W_i \cdot [x_t, h_{t-1}] + b_i) = \mathcal{O}(mn + m^2 + m) \tag{4.16}$$

Therefore the complexity of the proactive component can be written as:

$$Complexity_{time}(proactive) = \mathcal{O}(mn + m^2 + m)$$

$$\Rightarrow Complexity_{time}(proactive) = \mathcal{O}(m \times (m + n + 1))$$

$$\Rightarrow Complexity_{time}(proactive) = \mathcal{O}(N^2) \tag{4.17}$$

This is the value of one training epoch. For an LSTM training process with τ training epochs, the complexity becomes $\mathcal{O}(\tau N^2)$. However, it is important to note that the value of τ is a constant, thus the final proactive complexity can be reduced back to $\mathcal{O}(N^2)$.

Combining the time complexities of all three hybrid components, the final complexity for the hybrid autoscaler $\mathcal{H}$ can be re-written as:

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

$$\Rightarrow Complexity_{time}(\mathcal{H}) = \mathcal{O}(N^2) \tag{4.18}$$

From the results in Equation 4.18, it is clear that the hybrid algorithm performs extremely well in a polynomial time complexity. Even though it is shown that no mathematically certain decision can be computed in a polynomial time due to the NP-Hard nature of the problem statement, the hybrid algorithm is proven to provide an alternative that is able to approximate a high accuracy auto-scaling decision in polynomial time.

Chapter 5

System Implementation

Based on the previous high-level overview of the architectures and algorithms involved, in this chapter we will first discuss the micro-service application architecture used to test the auto-scaling on, along with its features in Section 5.1. With this background, Section 5.2 will give an explanation of the workload generator bundled along with the micro-service deployment, and how that was used to generate the time-series workload typically seen in edge deployments. Finally, we will discuss the technical configurations of the hybrid autoscaler itself in Section 5.3. This will involve how the workload information is communicated between cloud and edge layers, how the autoscaler parameters for both reactive and proactive subsystems are configured, and how the forecaster is configured to use the time-series data to generate forecast values for a sufficiently large time-period to minimize training times even further. Finally, the hyper-parameter tuning configured by the autoscaler controller will be further elaborated.

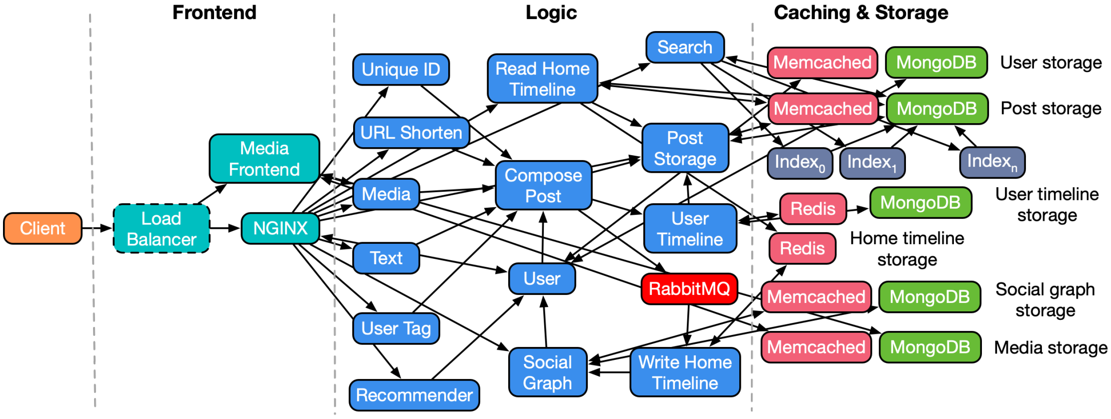
5.1 Micro-service Overview

The container orchestration used for deploying the micro-service deployment, custom hybrid autoscaler, as well as all the additional telemtry tools was Kubernetes. Kubernetes offered the highest configurability for such a project, and combined with its ease-of-use, made it ideal for stress testing auto-scaling. DeathStarBench [1], a social network micro-service implementation developed by Y. Gan *et al.* [90] was purpose-built

[1] https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork

for conducting benchmarks on edge architectures with constraints such as SLA agreements, and was thus deployed on the Kubernetes platform. The application mimics a typical large-scale social network application and supports common actions such as registering and login for user credentials, creating user posts on their timeline, reading other user timelines, receiving follower recommendations, following and unfollowing of other users, and searching for users or posts. Figure 5.1 shows the detailed architecture of the micro-service.

FIGURE 5.1: Social-Network architecture, source: [90]



The end-to-end service implementation depicts a social network with a broadcast style approach. The user or client sends requests using HTTP to the front-end layer. These requests are processed using NGINX [2], which is an open-source load balancer implementation. NGINX then specifies the web server that has been selected, and communicates with the micro-services in the logic layer, which are responsible for composing and displaying the user and timeline posts. The logic layer also consists of micro-services for serving advertisements, search engines, and other capabilities commonly found in large scale applications. The search and advertisement engines are machine learning plugins capable of serving recommendations based on user interactions. The logic layer is capable of handling posts containing text, links, and media. Users can also mark other user's posts as favourites, re-post them, and reply to them privately or publicly. Users can also follow, unfollow, or block others. All these interaction results are stored in the storage layer, which uses memcached for caching results, and MongoDB for storing user profiles, posts, media, and recommendations in a persistent manner.

[2] https://nginx.org/en/

Users can sign in to the application website by connecting to the user interface deployment, which can be assigned a DNS address. For simplicity, in this experiment the DNS configuration was skipped in favour of exposing the Kubernetes cluster IP address of the front-end deployment, along with the social media application port. For example, an API request for reading user timelines will look as follows:

```
http://<app-IP>:<app-port>/wrk2-api/user-timeline/read
```

5.2 Data Generation

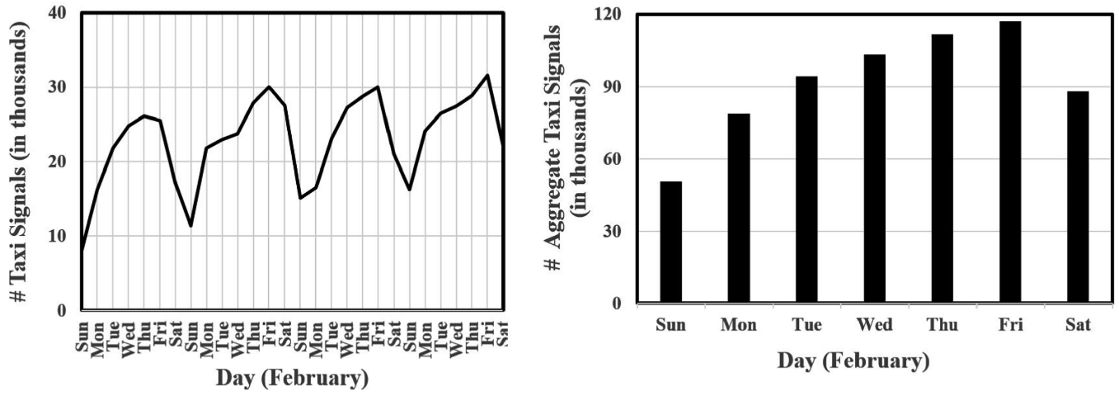
The social media deployment also comes with an HTTP workload generator, which was leveraged to create a realistic simulation of a typical day of workload for the microservice application. The generator, known as *wrk2*, is an open-loop load generator. HTTP requests are sent out according to the user configuration, regardless of whether or not the responses of the previous requests have been received. This avoids issues such as queuing delays in the server, helping to maintain an even load throughout the generation process which aids in benchmark process. The workload generator supports a variety of load generation use-cases, as well as different APIs. For example, a POST request with a total load generation of 8 requests per second, using 2 parallel threads, with 5 open HTTP connections, and for a duration of 30 seconds looks as follows:

```
$ wrk2 -t2 -c5 -d30 -R8 http://<app-IP>:<app-port>/wrk2-api/post/compose
```

A typical IoT application in the edge has a semi-predictable workload pattern. What this means is that while the exact workloads may vary from day to day, the overall pattern of when the workloads peak and drop mostly remain constant from day to day. Tadakamalla and Menascé [91] demonstrated this through a survey of three popular IoT datasets. Their results demonstrated that IoT application workloads can be well approximated using a lognormal distribution. Furthermore, the authors also observed that the daily routines of the users greatly affected the workload patterns in the dataset. This information is used to inform our own workload generation, to inject predictable patterns in the hourly data.

In this experiment, it is assumed that the workload peaks in the morning and evening, while seeing moderate usage during the afternoon. The workload is lowest at night,

FIGURE 5.2: IoT data characteristics, source: [91]



maintaining a low usage throughout. The workload simulator was modified to introduce an element of randomness to mimic realistic weekly workloads, which can vary on occasions such as weekends and public holidays.

To achieve this, an IoT workload algorithm was created for leveraging the *wrk2* generator to achieve this time-series. This is explained in Algorithm 6. A constant light workload is set for night time consisting of six hours between 12:00am and 06:00 am, while different workloads for the rest of the 18 hours of the day are manually set, with peaks set in the morning and evening, and dips in the afternoon. The algorithm also contains a small random "offset" variable to depict the randomness inherent in the workload. This is depicted using the function $\mathcal{F}$, where $\mathcal{F}(0, 1)$ returns a random number between 0 and 1. This workload is then executed using the *wrk2* generator to apply the HTTP load on the micro-service.

Algorithm 6 IoT daily workload generation

```
Input: $offset, night\_workload, day\_workload$
Output: $workload$
  for $\eta_i \leftarrow night\_workload$ do
      $workload \leftarrow generate(\eta_i + \mathcal{F}(-offset, offset)$
  end for
  for $\delta_i \leftarrow day\_workload$ do
      $workload \leftarrow generate(\delta_i + \mathcal{F}(-offset, offset)$
  end for
  return $workload$
```

5.3 Hybrid Autoscaler Configuration

With the Kubernetes container orchestration now installed on the servers, and the micro-service social network deployment set up on the orchestration platform, the hybrid autoscaler could be configured on the orchestration tool to read data from both Kubernetes and the micro-service to scale resources.

The default Kubernetes autoscaler has built-in API services to communicate with the micro-service deployments, along with any other telemetry deployed on the platform. These include built-in default metrics such as CPU and memory usages of the deployment. However, the proposed hybrid autoscaler must have these services custom deployed. Furthermore, to keep a track of custom metrics such as the SLA latency of various micro-service deployment modules, these metrics need to be extracted from the social-network in a time-series compliant manner. Thus, before describing the autoscaler configuration, a brief explanation of how the micro-service metrics are exported to Kubernetes in a manner in which they can be consumed by all other systems is discussed. Once that is explained, and the deployment is shown, the technical configurations, along with code-snippets of the various autoscaler subsystems can be laid out and discussed thoroughly.

5.3.1 Exporting Custom Metrics to Kubernetes

The social media application comes bundled with a deployment known as Jaeger [3]. Jaeger is an open-source distributed tracing platform, capable of tracking several application metrics of each network request and per-micro-service request. It then stores these metrics in a centralized database. One such metric which is critical for the functioning of the autoscaler is the details of API latency for the application. The latency information is extremely detailed, and is broken down per each component in the social media architecture layers as defined by Figure 5.1. Alongside the latency breakdown, Jaeger can automatically generate graphs of the various components which are utilized to serve the API requests. This is particularly useful for identifying deployment bottlenecks in the architecture which are prime targets for auto-scaling.

[3] https://www.jaegertracing.io/

Before this latency information can be useful in the auto-scaling solution, it must be imported to Kubernetes in a readable format. This was achieved through the use of a tool called Prometheus. Prometheus [4] is an open-source deployment used for monitoring micro-services and their applications. Prometheus consists of a multi-dimensional data model for storing time-series data through the use of key/value pairs, a custom built querying language known as PromQL which is used to leverage and search through this data model, and a graphing and dashboard user interface to aid in visualization. Prometheus allows for complex queries to be run on the real-time data, however due to the resource intensive nature of the deployment, Prometheus must be deployed on the cloud layer. That is the approach this experiment took, and Prometheus acted as the central database from which the autoscaler controller would periodically scrape social network metric data from for use by the auto-scaling components.

Listing 5.1: *Jaeger-Scraper* implementation to retrieve jaeger metrics

```
const avg_counter = new Gauge({
        name: 'avg_latency',
        help: 'Jaeger average post API latency (ms)'
});

get('/metrics', async (req, res) => {
    let url = process.env.JAEGER_URL;
    const data = await axios.get(url);
    let avg_duration = 0;

    let durations = data.map(components => {
        let duration = components.duration;
        return duration;
    });

    avg_duration = durations.reduce( (a,b) => a+b ) / durations.length;
    avg_counter.set(avg_duration);

    result.set('Content-Type', avg_counter);
});
```

To facilitate the export of Jaeger metrics to Prometheus, a custom deployment was created which scrapes these metrics at periodic and frequent time intervals, and pushes it to Prometheus. The deployment, which was named *Jaeger-Scraper*, was a JavaScript express server set up on Kubernetes using Docker, and the server code was implemented

[4] https://prometheus.io/docs/introduction/overview/

as shown in Listing 5.1. The server uses the open source NPM library *prom-client*[5] to create a Prometheus Gauge. A gauge is an extension of the Prometheus metric counter, the primary difference being that it can be both increased or decreased, whereas the counter can only be incremented. When Kubernetes requests metric data from the Jaeger-Scraper API endpoint, the scraper gets the latest Jaeger metrics within a fixed window in milliseconds (ms), and calculates the average latency of the time period. This value is then ready to be pushed to the Prometheus database.

Once this server is deployed on the Kubernetes cloud layer, a service-monitor tool for Prometheus is written, which tells Kubernetes to invoke the GET request to the API this server exposes at a set interval of time. It is through this API call that the latency value gets pushed to the Prometheus database. By default, the service-monitor calls the `/metrics` endpoint, which is what the Jaeger-Scraper is configured with. For this thesis, this interval was set at 15 seconds, and the configuration is shown in Listing 5.2.

Listing 5.2: *Jaeger-Scraper* service monitor for Prometheus

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app: jaeger-scraper
  name: jaeger-scraper-svc-monitor
spec:
  endpoints:
  - interval: 15s
    port: http
  selector:
    matchLabels:
      app: jaeger-scraper
```

With the service-monitor deployed, the scraped values were now visible when querying the metrics endpoint, and Listing 5.3 below shows an example of how the metrics were displayed. In a similar manner, these metrics could easily be queried on the Prometheus user interface using PromQL. However, while the latency metrics were now present in Prometheus, the next step was to import these metrics to the Kubernetes custom metrics server.

[5] https://github.com/siimon/prom-client

Listing 5.3: *Jaeger-Scraper* metrics collector example

```
$ curl $(kubectl get service jaeger-scraper --template \
'{{.spec.clusterIP}}'):8081/metrics
# HELP avg_latency Jaeger average post API latency (ms)
# TYPE avg_latency gauge
avg_latency 42.0
```

As mentioned above, the Kubernetes autoscaler is able to scale resources based on CPU and memory utilization. However, for more complex use-cases, more metrics need to be taken into account to make such decisions. To aid in this process, the Prometheus Adapter [6] was created to leverage the metrics collected and stored by the Prometheus deployment, and feed them to Kubernetes. These metrics were exposed via an API service and were consumed by the hybrid autoscaler for decision making.

Listing 5.4: Prometheus adapter configmap for avg_latency metric

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-metrics-prometheus-adapter
data:
  config.yaml: |
    rules:
    - seriesQuery: avg_latency{namespace!=""}
      resources:
        template: <<.Resource>>
      name:
        as: ${1}
        matches: ^(.*)
      metricsQuery: <<.Series>>
```

The prometheus adapter requires a configuration map (ConfigMap) to translate Prometheus metrics to Kubernetes custom metrics. The adapter does so in four steps.

- **Discovery**: The adapter discovers the metrics available in Prometheus.
- **Association**: It then figures out the association between each metric and Kubernetes resource.

[6] https://github.com/kubernetes-sigs/prometheus-adapter

- **Naming**: A name is then assigned to these resources through which the custom metrics API can expose them.
- **Querying**: Finally, the Prometheus deployment is queried to get the actual metric values.

The hybrid autoscaler requires the default CPU metric, as well as the custom latency metric. Thus we define this additional metric for configuring in Kubernetes. Listing 5.4 shows the discovery, association, naming, and querying steps to extract the average API latency. In the code snippet, the *seriesQuery* corresponds to discovery, *resources* to association, *name* to naming, and *metricsQuery* to querying.

Now, the Kubernetes custom metrics API exposes the following additional APIs under the resources pods, services and namespaces as shown in Listing 5.5. Alongside this, querying individual metrics gives resource values which will be used for autoscaling purposes.

5.3.2 Reactive Autoscaler

With the custom metrics now being exposed by Kubernetes, all the tools required for the reactive auto-scaling subsystem were in place. This would be built as an extension of the default Kubernetes horizontal pod autoscaler.

As discussed in Section 4.2.1, the reactive autoscaler cool-down parameters must be set to a value that is not too small or large. For this experiment, the parameters of the autoscaler were modified by setting both the scale up and scale down cooldown values to 15 seconds (equivalent to one control loop for the autoscaler controller), and maintain the toleration value at the default of 0.1. This means that if the controller sees that the autoscaler is requesting resource scaling for two consecutive control loops, it will proceed with the auto-scaling procedure. This ensures that reactive auto-scaling occurs as quickly as possible, to minimize SLA violations, while decreasing the chances of resources constantly being scaled up and down in an oscillating manner [92], which leads to unavailability of resources and drives up resource costs. Listing 5.6 shows the configuration snippet of the reactive autoscaler provisioning tool to achieve this.

Listing 5.5: Custom metrics API example

```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1 | jq .
{
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "services/avg_latency",
      ...
    },
    {
      "name": "pods/avg_latency",
      ...
    },
    {
      "name": "namespaces/avg_latency",
      ...
    }
  ]
}
$ kubectl get --raw \
/apis/custom.metrics.k8s.io/v1beta1/namespaces/default\
/services/*/avg_latency | jq .
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "items": [
    {
      "metricName": "avg_latency",
      "value": "42.0"
      ...
    }
  ]
}
```

While these parameters are configurable by other users, experimental results showed that for SLA-compliant applications, these parameters were capable of producing the best results for a wide array of use-cases. Higher cooldown windows lead to SLA-violations due to the time taken to scale resources, while smaller windows caused the constant scaling up and down of resources on small variations, leading to a loss of resource availability and increased deployment costs for the entire architecture.

Listing 5.6: Reactive autoscaler cooldown configuration

```
spec:
  behavior:
    scaleDown:
      policies:
      - periodSeconds: 15
        type: Pods
      stabilizationWindowSeconds: 15
    scaleUp:
      policies:
      - periodSeconds: 15
        type: Pods
      stabilizationWindowSeconds: 15
```

5.3.3 Proactive Autoscaler

The proactive autoscaler resource provisioning subsystem has a Kubernetes configuration that is similar to the reactive one shown in Listing 5.6. The same cooldown values of 15 seconds are applied here as well, to keep it consistent with the reactive solution.

The proactive Algorithm 3 was deployed on Kubernetes using the same method as the Jaeger-Scraper, and was accessible in the edge network by invoking a GET request to the API. However, to capture the $metric_{forecast}$ value returned by the forecaster in the custom API, the Prometheus Adapter configuration map required to be modified to append the value, as shown in Listing 5.7. In this experiment, the $metric_{forecast}$ value was represented by the variable $forecasted\_cpu$. With this addition, the proactive autoscaler was able to receive predicted CPU workloads.

Listing 5.7: Prometheus adapter configmap for forecasted_cpu metric

```
- metricsQuery: <<.Series>>
  name:
    as: ${1}
    matches: ^(.*)
  resources:
    template: <<.Resource>>
  seriesQuery: forecasted_cpu{namespace!=""}
```

The proactive forecaster was a deep-learning machine learning model configured with a multi step forecast output as discussed in Section 2.4.2. The model consisted of three

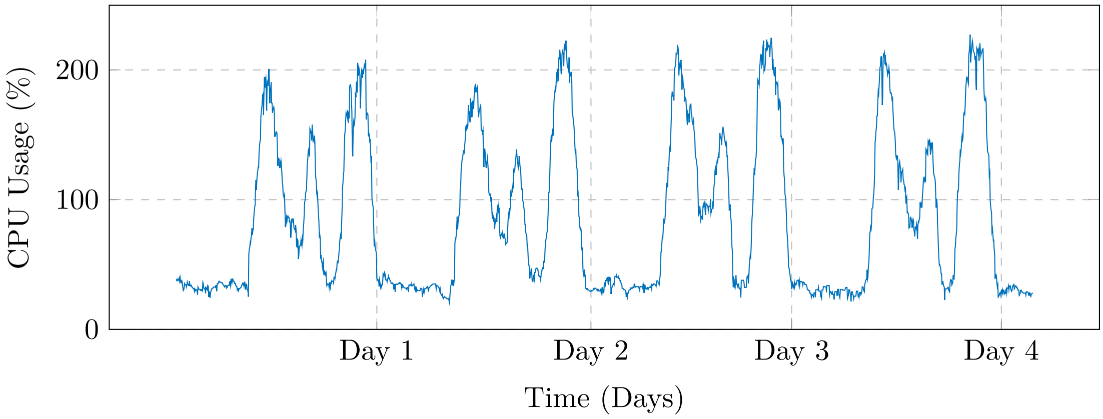
Table 5.1: Overview of proactive forecaster layers.

Layer Details	**Output Shape**	**Parameter Count**
$LSTM_1$	(10, 50)	10400
$Dropout_1$	(10, 50)	0
$LSTM_2$	(10, 50)	20200
$Dropout_2$	(10, 50)	0
$LSTM_3$	(50)	20200
$Dropout_3$	(50)	0
$Dense_1$	(540)	27540

LSTM layers, alternated with two dropout layers. These dropout layers were used to prevent over-fitting of data during the training process. Finally, the last layer was a densely connected neural-network which generated the forecaster output in the required shape. For this experiment, the output was 540 data points, which was approximately the amount of data required to forecast 24 hours of workload. By doing so, the forecaster only needed to be run once a day, vastly reducing total training time. This was only possible due to the data pre-processing done before training. Table 5.1 shows the detailed information about the forecaster model layers.

The step-by-step forecast procedure was done as follows. The data which was generated by the workload Algorithm 6 and stored in Prometheus was periodically queried and stored by the autoscaler controller. An example of how this data may look over a period of four days is shown in Figure 5.3.

Figure 5.3: Example of generated historical workload



This data has two major peaks during the morning and evening, with one smaller peak in the afternoon. The night time workload is consistently minimal. Due to the randomness included in the algorithm, each of the peaks are never the exact same, which helps mimic

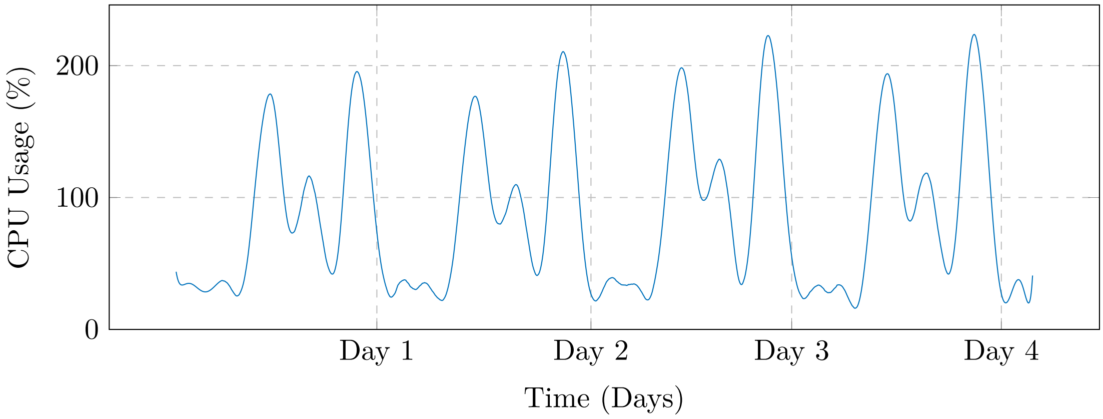
TABLE 5.2: Proactive forecaster hyper-parameter values.

Hyper-parameter	**Value**
learning_rate	0.005
epochs	75
batch_size	100
optimizer	Adam

the real data an edge architecture would experience. However, this data has several abrupt changes every few minutes, for example the utilization could be 100% at 5:00pm, then suddenly drop down to 75% at 5:10pm, before coming back up to 110% at 5:20pm. These abrupt changes makes it difficult for the LSTM to be able to accurately predict the future workloads, and requires a much larger input window sequence to reduce model loss, which can significantly drive up training times to an unfeasible amount.

To get around this issue, the proactive autoscaler introduced a data pre-processing component. This involved the use of noise reduction and data smoothing algorithm. This was done through a popular algorithm known as the Savitzky-Golay filter [93]. This filter takes N points in a given time-series, with a filter width w, and calculates a polynomial average of order o [94]. The resulting time-series data which can be seen in Figure 5.4 has considerably less deviations between consecutive points, and devoid of noise.

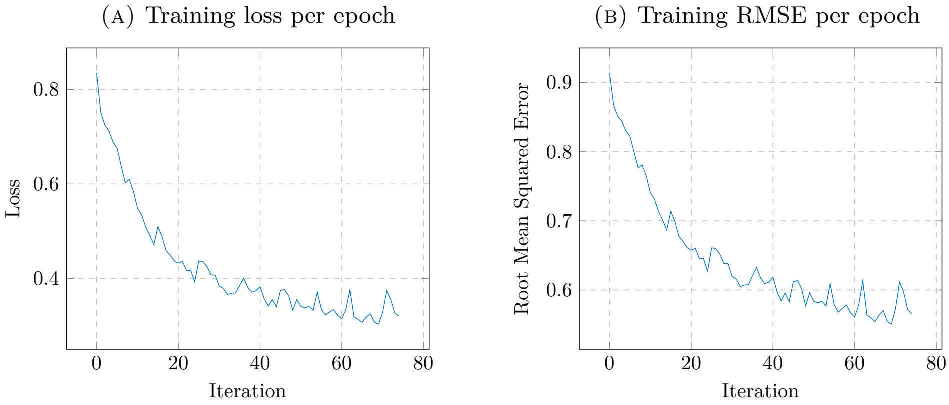
FIGURE 5.4: Example of pre-processed historical workload



The filtered data was then normalized to further remove the impact of scale between different applications, which helps generalize the autoscaler. The data was then ready to be used to train the model. Alongside the architecture configured in Table 5.1, the LSTM model contained the following default hyper-parameter values depicted in Table 5.2.

It was experimentally discovered that a learning rate higher than the configured value reduced the prediction accuracy drastically, as the model was unable to find the optimal loss value during training, becoming stuck in a local minima. Furthermore, attempting to alleviate this issue by increasing training epochs to over 200 resulted in severe over-fitting, devaluing the algorithm for generalized workloads. Thus the hyper-parameters were chosen in such a way that they minimized training time and over-fitting, while maximizing prediction accuracy within the constraints of limited edge layer resources. To further reduce over-fitting, an additional function known as *early-stop* was defined which halted the training of the model if the loss did not decrease for 10 consecutive training epoch iterations. Lastly, an Adaptive Moment Estimation (Adam) optimizer [95] was used in an attempt to generate the highest accuracy in the least amount of time. These parameters were determined based on the research experiments conducted by Siami-Namini *et al.* [89].

FIGURE 5.5: Analysis of loss and RMSE in training

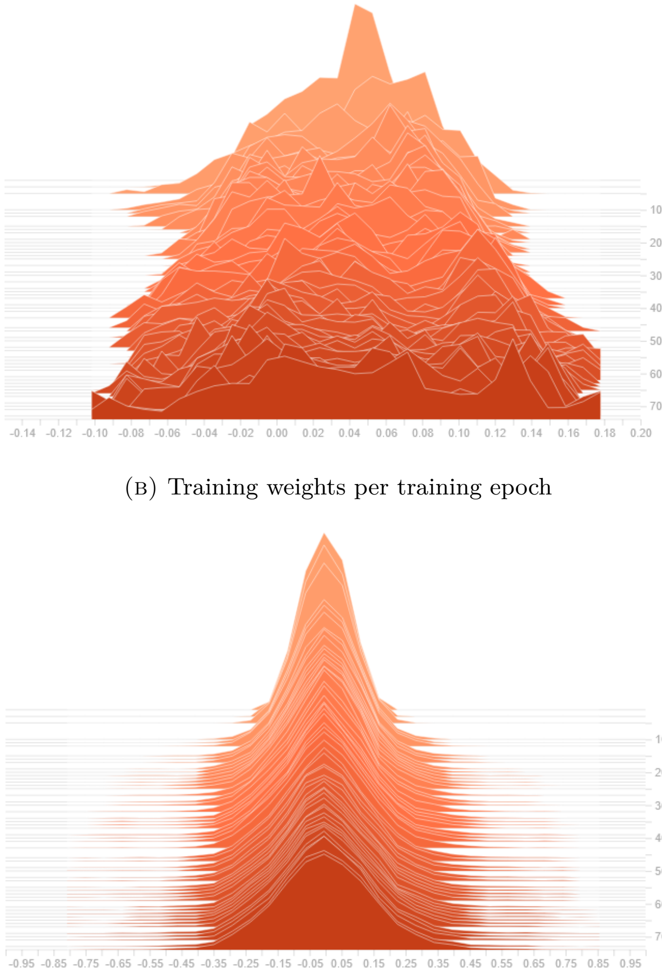

(A) Training loss per epoch

(B) Training RMSE per epoch

Figures 5.5A and 5.5B show how the loss and root mean squared error of the model decreased with respect to the training epochs. The loss dropped drastically for the first 25 epochs, before reducing gradually until the 65^{th} epoch. From there on until the end of training, the loss value plateaued, and even spiked in certain epochs, making the limited gains too costly in terms of training time. From this, the choice of 75 epochs for the model training was justified.

A similar trend was seen in the weights and biases of the LSTM. In the first epoch, the biases were uniformly distributed, but slowly converge to a smaller value. Similarly, the dense layer weights were able to converge to the values with least loss in the final epoch. Figures 5.6A and 5.6B show how these values for the final dense layer changed over each epoch.

Figure 5.6: Analysis of bias and weights in training

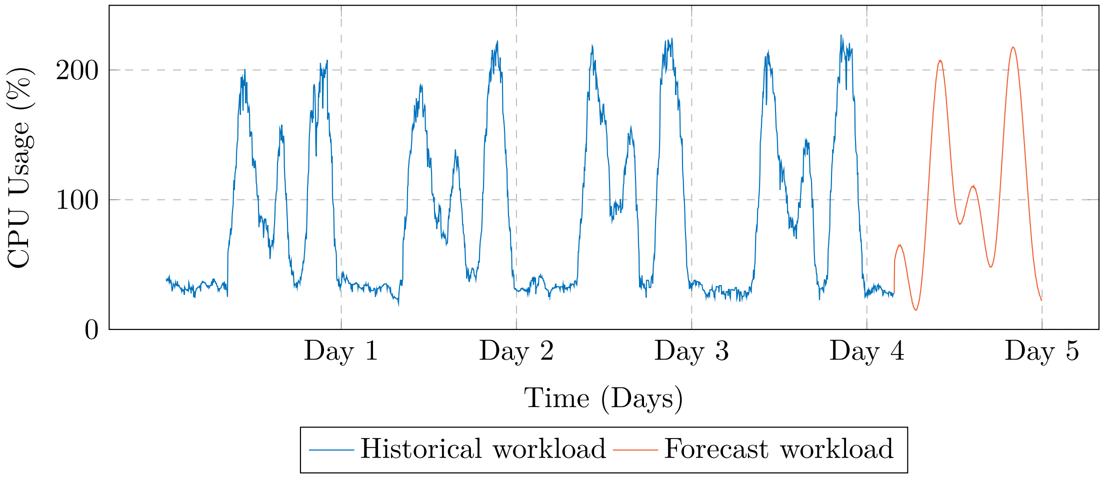


(a) Training bias per epoch

(b) Training weights per training epoch

The model training, validation and error comparison with previous model performances took approximately 3 minutes, after which the model predicted the subsequent day's forecast in under 10 seconds. Figure 5.7 shows how this looked. The forecaster accurately depicted the initial peaks for the entire day, including the morning, afternoon, and evening. It is clear however, that the rest of the data points may not be as accurate. This is not a significant drawback for the hybrid model, as the reactive algorithm is capable of making minor adjustments to the resources, ensuring that the SLA compliance is maintained.

Figure 5.7: Historical workload paired with forecast workload



5.3.4 Autoscaler Controller

The autoscaler controller stored a maximum of seven days of data for use by the proactive forecaster. Data that is too old is not particularly useful for a time-series model training on semi-predictable data, and only increases training time without adding much to the accuracy [96]. This data was refreshed once every day to ensure that the latest data was always available to the hybrid autoscalers before training. Since training only took place once a day, the costly metric scraping operations done by the controller from the cloud layer could be reduced, thereby reducing the overall load on the network.

The controller sent a training request to the proactive autoscaler once every night. The time is chosen so that, due to the comparatively low amounts of user requests, the training process would be able to claim as much of the edge resources as possible without affecting the network availability. However, before sending the request, the controller computed whether or not an SLA violation took place in the past 24 hours. If it detected a violation, it assumed that the forecaster had not learnt enough of the time-series features to make an accurate prediction. This could be due to a variety of reasons such as lack of training data or conservative parameter values. To counteract this, the controller performed the following corrections:

- *learning_rate* was decreased by 0.0005, to a minimum value of 0.002.
- *epochs* was increased by 5, to a maximum of 100.
- *batch_size* was increased by 10, to a maximum of 200.
- *early_stop* threshold was increased by 5, to a maximum of 25.

The assumption here was that the training accuracy could be jump started by sacrificing short training times for one duration to better learn the time series features and thus reduce SLA violations. As long as SLA violations occur, these modifications keep being performed up until their configured limits. Experimental testing demonstrated that, at the most extreme hyper-parameter configuration, the training time increased to 10 minutes. Once no SLA violations were detected during a 24 hour time window, the hyper-parameters were reverted back to the default values. Doing so did not cause the model to lose any previous optimizations or reduce prediction accuracy, but had the added benefit of reducing training time and resources once again.

Chapter 6

Performance Evaluation

In this chapter, we begin by discussing the underlying server hardware configuration, and the assumptions made before beginning the auto-scaling experiments in Section 6.1. The cluster configuration, which involves the resource divisions between servers, overall cluster architecture, and deployment resources is discussed in Section 6.2. The experimental setup which details the SLA categories, its latency values, and an overview of the two experiments to be conducted are discussed in Section 6.3. After this, Section 6.4 contains an overview of the three baseline algorithms which were used to compare the performance of the proposed hybrid autoscaler. Section 6.5 follows this with an overview of the workload which was generated to test the two experimental conditions, along with a brief analysis of the latency graph. Finally, the experimental results are shown and discussed in Sections 6.6 to 6.8.

6.1 Assumptions and Virtual Machines Setup

For the underlying virtual machine (VM) setup, servers in the Melbourne Research Cloud [1] were leveraged to deploy micro-services on. 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 a substantially higher amount of CPU cores and memory assigned compared to the servers in the edge layer, to simulate the scarcity of resources in the edge layer. The cloud layer also contained a

[1] https://docs.cloud.unimelb.edu.au/

200GB persistent storage volume, while the edge layer did not. This allowed the cloud layer to hold all the Prometheus data, while the edge layer had to store the time series in the RAM to depict the scarcity of resources further. Finally, a simulated latency was added between inter-layer server communication to mimic the perceived distance between edge nodes and large data centres.

Each server consists of an Ubuntu 22.04 [2] operating system. Kubernetes v1.28.2 [3] is used as the container orchestration technology behind the experimental micro service setup. CRI-O was installed on the Kubernetes nodes as the container runtime of choice for running resource nodes. Finally, for inter-pod communication on the cluster to occur, an open-source project known as Flannel [4] was deployed.

For maximum flexibility in modifying and monitoring the underlying architecture, a bare-metal implementation of Kubernetes was used, instead of ready-made solutions available from Amazon or Google. The control plane was deployed on the cloud layer, while the data plane was on the edge layer. Furthermore, several assumptions were made before proceeding with the experimentation:

- The only auto-scaling performed would be horizontal pod auto-scaling. Vertical and cluster auto-scaling were out of the scope of this project.
- The pods on which auto-scaling is not applied would have the maximum possible resource allocation, so as to remove the chances of bottleneck.
- At no point in the experiment would a node be taken down, or new node be added.
- The autoscaler assumes that every node in the edge layer is an equally likely candidate for scheduling pods on.

6.2 Cluster Configurations

The VMs were divided into the cloud and edge layer as depicted in Table 6.1. The control plane was further divided into two servers, one for handling the Kubernetes control plane scheduling, API service, and etcd deployments, and the other for storing

[2] https://releases.ubuntu.com/jammy/
[3] https://kubernetes.io/releases/
[4] https://github.com/flannel-io/flannel

Table 6.1: Cluster architectural layout

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

the Prometheus database, along with the micro-service Jaeger metrics collection. The edge layer consisted of four servers with far substantially fewer resources, to depict the difference in computing power. The network layer between the edge and cloud deployments also contained a simulated latency to denote the perceived geographical distance between them.

The cloud and edge nodes were differentiated through Kubernetes through the internal labeling system. A key *type* with the value either *cloud* or *edge* was added to each node. This would enable the scheduler to consider restricting the deployment of pods to particular nodes automatically. For example, the Prometheus deployment would only be deployed on the node of the type cloud. This process is known as *node affinity* [97].

There were several options to deploy the social media application to Kubernetes. Manually cloning them from the repository, creating the custom resource definitions (CRDs), and deploying the YAML files was an option that gave maximum flexibility, but was difficult to debug if things went wrong. Due to this, the Kubernetes package manager Helm was used. Helm [5] is another open-source project whose primary goal involves streamlining the installation, maintenance, and removal of Kubernetes deployments. This is achieved through the use of a Helm chart, which details the configuration of the project, and how to update and access it.

Therefore, the social media application was the first to be deployed on Kubernetes. However, before deploying the application, the primary deployments to be tested need to be configured. Based on the *wrk2* benchmark that was discussed above, two APIs were identified. One was a GET call to the user's home timeline, and the other was a POST request made by the user to create a post.

[5] https://helm.sh/

First, the social network with default configuration values was installed using the helm command below:

Listing 6.1: Social network installation using Helm

```
$ helm install social-media \
/DeathStarBench/socialNetwork/helm-chart/Chart.yaml -n default
```

With the social media network now deployed, a workload for both GET and POST requests was invoked to generate the latency trace on Jaeger, as depicted in Listing 6.2.

Listing 6.2: Generate Jaeger trace

```
$ WRK2="/DeathStarBench/wrk2/wrk -D exp -t 1 -c 1 -d 1 -L -s -R 1"
$ HT=./wrk2/scripts/social-network/read-home-timeline.lua
$ CS=./wrk2/scripts/social-network/compose-post.lua
$ IP=$(kubectl get svc nginx-thrift --template '{{.spec.clusterIP}}')
$ PORT=8080
$ $WRK2 $HT http://$IP:$PORT/wrk2-api/home-timeline/read
$ $WRK2 $HT http://$IP:$PORT/wrk2-api/post/compose
```

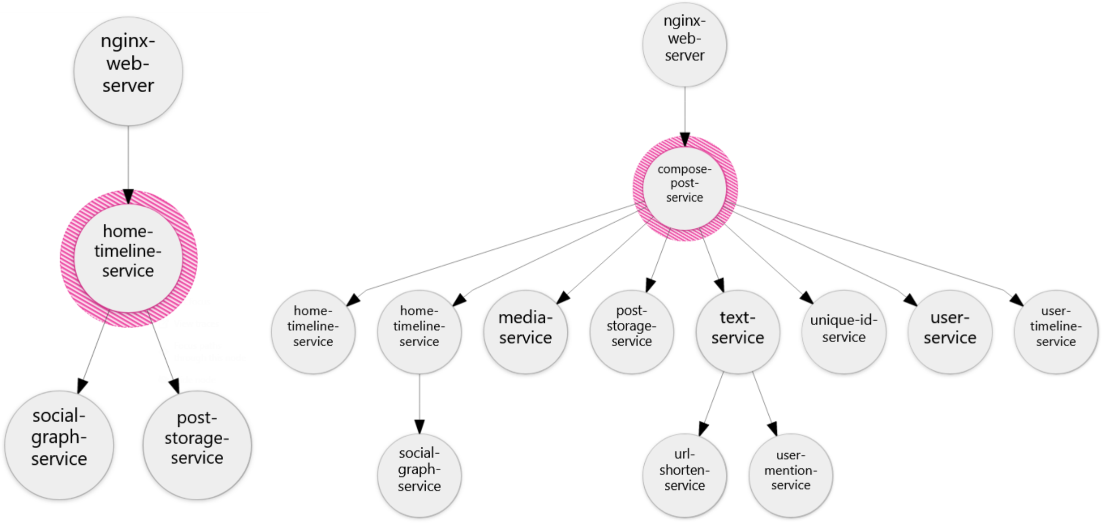
With the traces generated, the deployments that require auto-scaling could be identified for each. From the Jaeger traces generated in Figure 6.1, it was clear that the two deployments highlighted in pink, namely *home-timeline-service*, and *compose-post-service*, were the major bottlenecks in GET and POST request processing respectively, having to serve multiple child deployments simultaneously. This made them the most important deployments for auto-scaling. Therefore, the helm deployment was updated to assign auto-scaling resource limits to them. The resources were assigned in a realistic manner consistent with edge deployments and based on the number of components each deployment answered to. Listing 6.3 shows the CPU resources assigned to both deployments.

Listing 6.3: Update resources for bottlenecked deployments

```
    $ helm upgrade social-media \
    /DeathStarBench/socialNetwork/helm-chart/Chart.yaml -n default \
    --set-string home-timeline-service.container.resources="requests:
          cpu: "15m"
        limits:
          cpu: "15m"" \
    --set-string compose-post-service.container.resources="requests:
          cpu: "30m"
        limits:
          cpu: "30m""

```

Figure 6.1: API trace for *home-timeline-service* and *compose-post-service*



6.3 Experiment Setup

Two independent experiments were conducted to validate the performance of the hybrid autoscaler. The social media application was first tested using the GET request to the API to autoscale the *home-timeline-service* deployment. Then, a more demanding as well as challenging workload was applied to the POST request for auto-scaling the *compose-post-service* deployment. For both these experiments, the workload generation algorithm was used to create realistic daily workloads and tested over a period of five days.

According to Nilsson and Yngwe [98], their research concluded that user experience was negatively affected by higher API latency. This leads to a cascading issue of lower

Table 6.2: Experimental SLA constraints

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

revenue growth and profit. Their research found three latency brackets where the user experience changes.

- A response time of at least *100 milliseconds* was considered by the user to be instantaneous. This is the best case scenario.
- A response time of up to *1 second* was noticed by the user as a slight delay, however their user experience was not significantly interrupted.
- A response time exceeding *10 seconds* was deemed to cause the user to lose focus, and thus disrupted their user experience.

Based on this research, it was determined that for both the experiments, the flexible, moderate, and strict SLA constraints explained in Section 4.1.1 would be bound within the values $SLA(c) \in [100ms, 1000ms]$.

The flexible SLA constraint was the primary focus of the experiment. This threshold was chosen as it is the most common threshold used for IoT applications, and thus provide a good benchmark for application performance. However, a moderate and strict SLA constraint would also be chosen and tested to compute the SLA violation rates for the algorithm. The SLA values are shown in Table 6.2.

For the proposed hybrid algorithm to achieve these auto-scaling goals within the SLA constraints, the auto-scaling subsystems were configured as follows. The reactive autoscaler would check if the CPU utilization of the deployment was exceeding the auto-scaling threshold. This threshold would vary for each of the experiments as required. If this threshold was breached, it would scale up based on the cooldowns and tolerations set. The proactive autoscaler on the other hand would check if the forecasted CPU utilization in the next 20 minutes was going to breach the auto-scaling threshold, and if so, would autoscale with the same configured parameters as the reactive one. The controller would store the total CPU utilization of the deployment as a time-series for a maximum of seven days, and constantly check for SLA violations to tweak the hyper-parameters of the forecaster as discussed above in Section 5.3.4.

6.4 Baseline Algorithms

To measure the effectiveness of the hybrid autoscaler, three baseline algorithms were chosen for comparison. All three would autoscale at the same CPU auto-scaling threshold to maintain consistency. Furthermore, these algorithms would be implementations based on the ones discussed in Sections 3.4 and 3.5.

1. **Default Kubernetes Horizontal Pod Autoscaler**: No modifications to the configuration are made, thus the scale-up cooldown is 0 seconds, while the scale-down is 300 seconds. Additionally, the autoscaler has no knowledge of the workload distribution or SLA violations on the edge nodes.

2. **Traffic Aware Horizontal Pod Autoscaler**: This is an implementation of the reactive THPA algorithm created by Phan *et al.* [39]. This autoscaler scheduler will compute the ratio of workloads being exerted on the different edge nodes with the deployment pods. Once it does so, it will scale these resources in a commensurate proportion.

3. **Proactive Pod Autoscaler**: This is an implementation of the PPA algorithm devised by Ju *et al.* [63]. This algorithm was an open-ended implementation that enabled the user to plugin a deep learning model of their choice. The PPA architecture consists of three sub-sections, the formulator, evaluator, and updater. An LSTM model is injected into the autoscaler as the model file. This LSTM implementation is similar to the one used in the hybrid autoscaler, however, it differs in two key elements. First, the LSTM does not expect pre-processed data without noise and thus deals with more complex time-series data. Secondly, due to this additional computation, the LSTM contains a deeper architecture layer with more neural network units. This is required as the algorithm has to correctly predict the complete future workload since there is no reactive autoscaler to fall back on. Over a fixed interval, the algorithm continuously loops through the time-series data and saves the forecast result to a metrics file. The evaluator takes these outputs from

the metrics file, along with the LSTM from the model file to predict the number of pods to assign in advance, and requests the Kubernetes scheduler for scaling through the API Service. A second loop, known as the update loop, then updates the LSTM model using the latest forecast, and clears the metrics file. The hyper-parameters are carefully tuned to ensure that the model does not underperform. Finally, the PPA architecture does not take into account SLA compliance, and thus SLA metrics are not provided as feedback for hyper-parameter tuning.

6.5 Experimental Workload

In the first experiment, the IoT workload generation algorithm was configured to create a workload aimed at mimicking GET requests. By doing so, the *home-timeline-service* deployment became the bottleneck receiving all of the requests and thus could be tested using the proposed hybrid autoscaler, as well as the three baseline algorithms. As explained above, the auto-scaling threshold for all four algorithms was set to 50% total CPU utilization on the deployment, and the SLA latency threshold was set to the flexible threshold of 150 milliseconds.

FIGURE 6.2: Total CPU usage for *home-timeline-service*

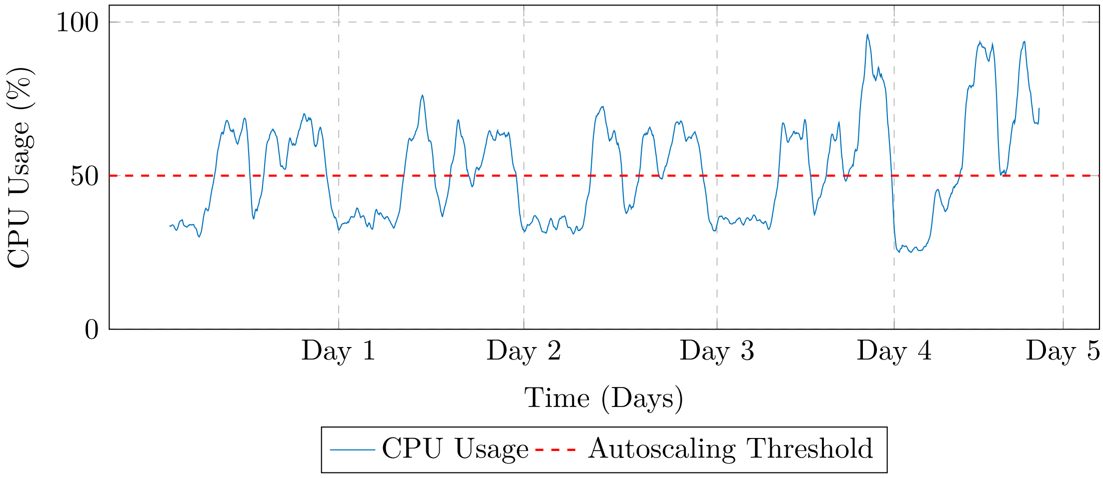
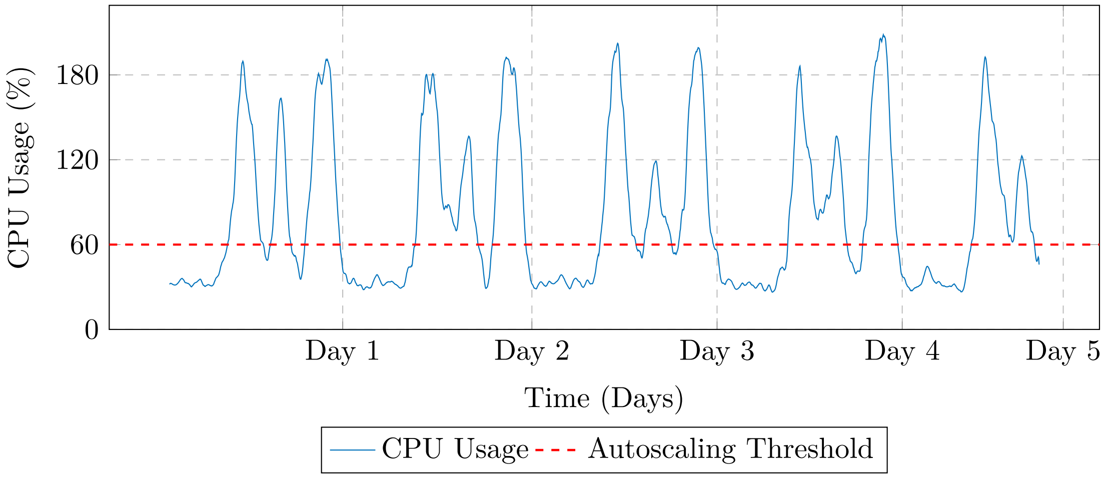


Figure 6.2 shows the total CPU workload that was generated by Algorithm 6 on the *home-timeline-service* deployment. The data was generated for a total of five days. Throughout the five days, approximately 2,550,000 requests in total were sent to the edge deployment. The daily peak remained at approximately the same percentage but then increased significantly in the last two days. This could be interpreted as a depiction of

how social network requests may look on the weekends. The total CPU utilization never exceeded 95% of the total allocated resources shown in Listing 6.3. This is expected of a GET request, as while the deployment has to open a connection to the sub-deployments to receive the response, a GET request is typically a database SELECT statement, which takes a comparatively less amount of time as opposed to other operations. This means that the deployment can quickly receive the response from its child components, send the response up to the requester, and then close the connection. Once the connection is closed, all the resources associated with it are freed. Because this operation is so quick, multiple concurrent connections are typically not open, and thus the CPU utilization is not significant.

For the second experiment, the focus was on a more difficult auto-scaling task. The IoT workload generation Algorithm 6 was reconfigured to simulate a realistic workload for social network users writing user timeline posts, which at a low level, involved sending POST requests to the edge deployment. Doing so meant that the *compose-post-service* would be sending all these POST requests to its sub-components for writing media or text. The workflow for a POST request to the API goes as follows. The user sends a POST request with the body containing the data to be written. This is sent to the *compose-post-service* deployment which opens an ephemeral connection with the user. Simultaneously while doing so, the deployment sends the contents of the POST body to one of its sub-components. These components house the database for user posts, and the operation is performed using an INSERT or UPDATE statement. The sub-component then sends its response back to the *compose-post-service* informing it whether or not this operation succeeded or failed. The deployment sends this information back up to the user and closes the connection.

Figure 6.3: Total CPU usage for *compose-post-service*



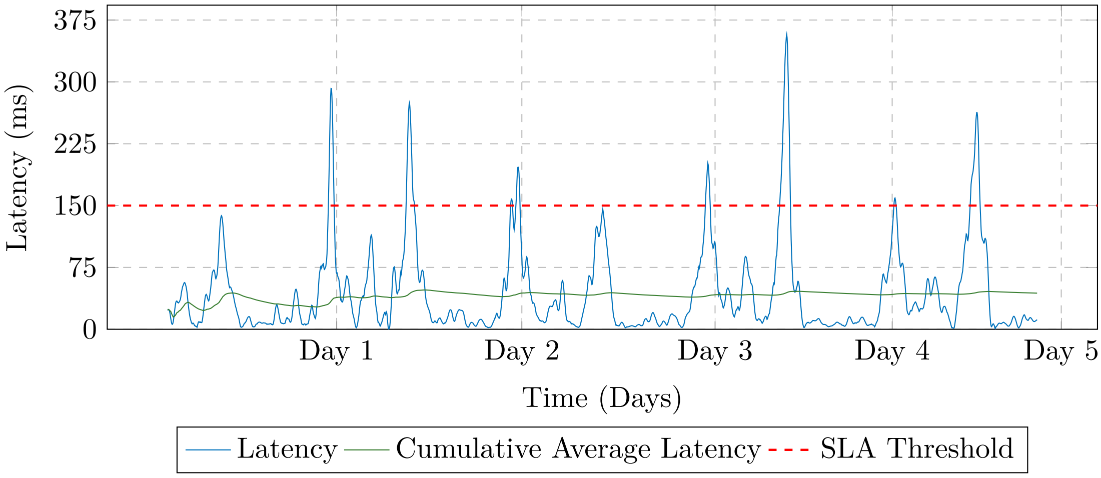
While the SELECT statement used underneath the GET request was a fairly quick operation, the INSERT and UPDATE statements may take significantly longer, due to the idempotent nature of the database [99]. The social network will ensure that user posts are in the correct order, and perform other background checks before committing the data. Due to this, the connections on the *compose-post-service* can remain open for a significantly longer amount of time, resulting in far more resources being used. Figure 6.3 shows this in action. The algorithm was generated for a total of 5 days, similar to the first experiment. Even though the workload generation algorithm was sending the same amount of POST requests as it did GET requests, approximately 2,550,000 requests in total, the total CPU workload in the second experiment peaked at approximately 175%, and at some points even reached 200% of the total CPU resources assigned to a unitary deployment pod. This varied per day in a realistic manner similar to what could commonly be seen in social networks.

To resolve this CPU bottleneck, the auto-scaling threshold was set to 60% of the CPU workload. In addition, the SLA latency threshold was set to the flexible threshold of 1000 milliseconds. Using these configurations, the three baseline algorithms along with the proposed hybrid autoscaler was tested.

6.6 Evaluation of Request Latency

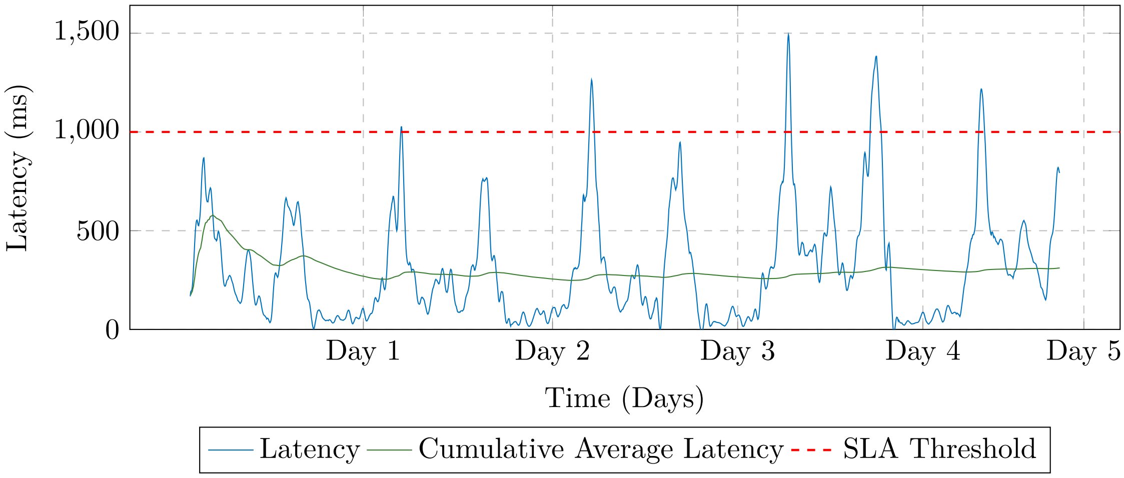
6.6.1 Default Kubernetes Autoscaler Baseline

FIGURE 6.4: Kubernetes default sutoscaler latency for *home-timeline-service*



For both the workloads discussed above, a near 100% CPU utilization on a solitary deployment pod would lead to significant delays. This can be seen when using the default Kubernetes Horizontal Pod Autoscaler in the first experiment, as depicted in Figure 6.4. The autoscaler was merely a primitive reactive implementation with no knowledge of which edge nodes were experiencing heavy traffic, and which ones were not. Thus, it blindly assigned pods to the nodes in a round-robin manner. Additionally, the autoscaler required a significant amount of time to register the new pods to the Kubernetes control plane, thus falling victim to the cold start problem. This results in significant latency spikes before the resources are adjusted. In the figure, it can be seen that the latency exceeds 300 milliseconds at some points, which would be significantly large enough to degrade the user experience. Additionally, by the end of the fifth day of testing, the cumulative average latency of the social-network was nearly 50 milliseconds.

FIGURE 6.5: Kubernetes default autoscaler latency for *compose-post-service*



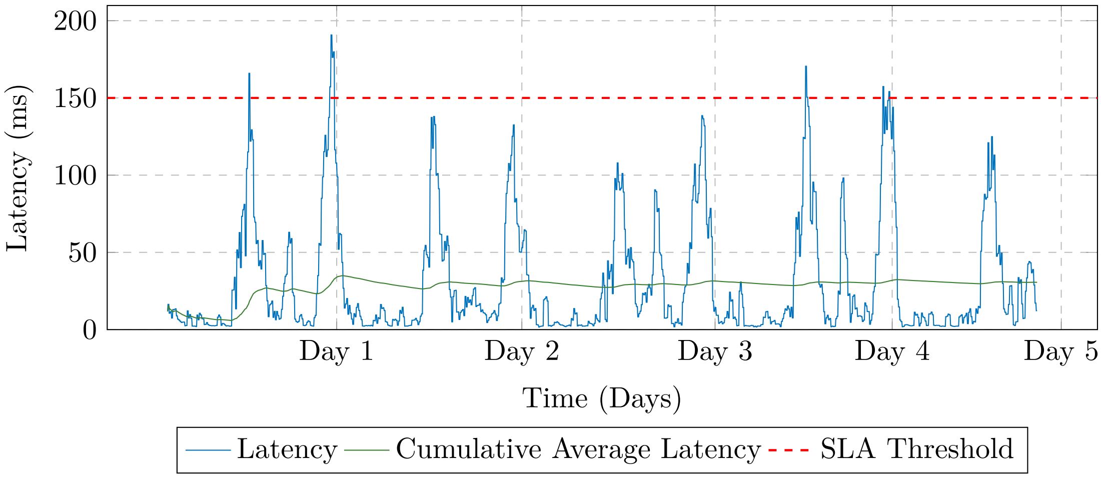
Similar to the results in the first experiment, the shortcomings of this auto-scaling approach were exposed even more so by the increased demands of the POST request workload in the second experiment. Figure 6.5 shows the latency metrics of the autoscaler. During the daily workload spikes, the autoscaler would regularly breach the 1000 millisecond threshold, with values peaking at almost 1450 milliseconds. This is more than 45% above the allowed threshold, which would cause general system instability. Furthermore, the average latency of the autoscaler hovered at around the 400-millisecond mark throughout the experiment, which was quite a high value when taking into account the large periods of non-peak workload in the test.

Upon further investigation, it was discovered that the high latency was caused due to three issues. One was the cold start problem, which added a constant latency to the non-proactive algorithms which could not be solved by any reactive solutions. The second one was the avalanche effect which was caused by resources not being available in a timely manner, and was related to the cold start. Before the Kubernetes control plane could register all resources, the *compose-post-service* deployment may have had a CPU utilization at or near 100%. When this happened, the deployment already had all available resources being utilized to open connections, this caused additional connection requests to be dropped. The user does not see this happen and instead waits the default 60 seconds before displaying a “time-out” message. This 60 seconds was taken into account when measuring the latency and was what drove up the latency to such an extent. Finally, the Kubernetes default autoscaler had no information on which edge nodes were receiving the most number of requests. It only saw CPU utilization metrics as a totality. As a result of this, it scheduled new resources in a round-robin manner, meaning that some deployment pods received a lot of requests, while others may have received none. This uneven distribution of requests to the resources also drove up latency.

Based on these experimental results which showed that connections were being dropped, it meant that the social network was becoming unavailable at certain points in time during the day. This happening during peak hours was especially critical and violated SLA constraints. Due to this, the default Kubernetes autoscaler could not be considered viable for auto-scaling an edge deployment.

6.6.2 Reactive THPA Autoscaler Baseline

Figure 6.6: THPA reactive autoscaler latency for *home-timeline-service*



Unlike the default Kubernetes autoscaler, THPA kept track of which edge node was receiving a significant number of requests and assigned pods to the nodes accordingly. This strategy resulted in a significantly improved latency graph for the first experiment, as can be seen in Figure 6.6.

While the algorithm still suffered from the cold start problem, the more intelligent assignment of resources resulted in fewer availability issues, ensuring that the latency spikes that were seen were not as drastic as the ones in the default implementation. However, due to the cold start problem, the SLA threshold of 150 millisecond was still regularly breached, resulting in several violations and degraded the user experience. However these breaches never exceeded the 200 millisecond mark, and thus the drop in user experience quality was not as noticeable as the approach using the default Kubernetes baseline. Furthermore, the average latency over the experimental time frame was lower than what was seen in the default implementation, hovering at around 25-30 milliseconds.

FIGURE 6.7: THPA reactive autoscaler latency for *compose-post-service*

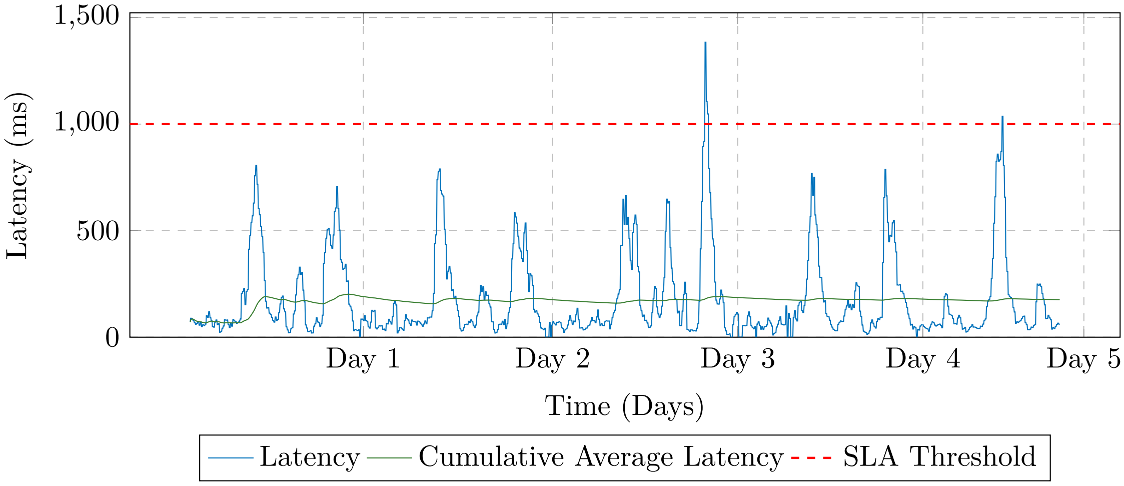


Figure 6.7 shows how the THPA algorithm performed on the POST request workload for the second experiment. As expected, the request-aware architecture of this autoscaler allowed it to eliminate the issues with dropped requests seen in the Kubernetes autoscaler implementation. This was due to the autoscaler assigning resource pods to the edge nodes which were experiencing a high number of POST requests. This ensured that on average, the latency values were kept lower. Furthermore, the avalanche effect seen above was somewhat mitigated due to the more intelligent resource deployment model.

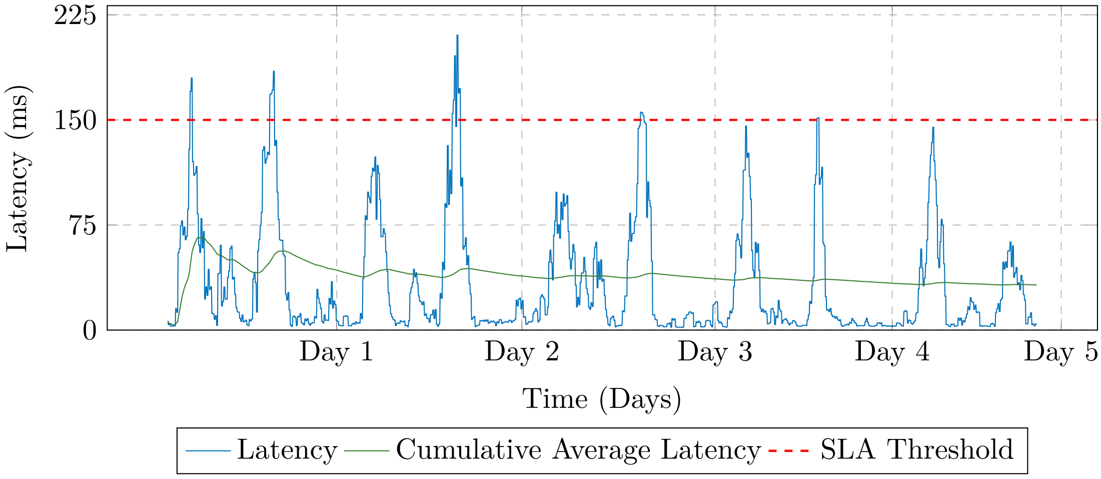
However, the cold start problem was not eliminated, which caused spikes in the latency when the autoscaler could not handle assigning resources in time. While this was less common in this implementation than in the default one, it nevertheless resulted in latency spiking above the SLA threshold for lengthy amounts of time on multiple occasions. On one occasion, the latency nearly hit 1400 milliseconds for several minutes before coming back down when the resource registration was completed.

These issues were not as severe as the ones seen in the default autoscaler however, and would not cause complete system unavailability. Furthermore, the cumulative average latency throughout the testing period was substantially lower than the default Kubernetes baseline, averaging at around 200 milliseconds. Despite this, SLA agreements were still being violated regularly, and thus while the autoscaler could be considered a suitable algorithm for native cloud deployments, it was deemed unsuitable for edge deployments with SLA constraints.

6.6.3 Proactive PPA Autoscaler Baseline

Unlike the previous two baseline algorithms, the Proactive Pod Autoscaler algorithm attempted to predict the workload before it was requested, thus eliminating the cold start issue. In ideal conditions, this would result in the SLA threshold not being violated, and it being a viable solution for edge paradigms. However, experimental results showed otherwise.

FIGURE 6.8: PPA proactive autoscaler latency for *home-timeline-service*



Because the autoscaler was purely proactive, it must be a deep LSTM model with several layers and large training epochs. This deep model took more than 50 minutes to properly train and validate for it to predict 24 hours of data, similar to the length of data that our proposed hybrid solution predicted. This is due to the edge architecture's lack of resources and storage compared to the cloud layer. Figure 6.8 shows this in action for the first experiment. Initially, it was observed that the latency continually spiked, causing a large amount of SLA violations, more than what the reactive autoscaler caused. However, after a few days of training, the rolling update structure of the LSTM weights took over, reducing the training time by taking advantage of the early stopping callback in the LSTM model, and managing to stabilize the latency. However, this effect did not always manage to stem the latency overflow, as was shown when the same algorithm was tested using the POST request in Section 6.6.3.

While the SLA violations were not as severe as the ones seen in the Default autoscaler baseline, they were comparatively greater than the reactive approach, with the latency exceeding 200 milliseconds for several minutes during the day, which would cause a noticeable delay. The average latency was almost as large as what was seen in the default autoscaler baseline, approaching 50 milliseconds due to the issues inherent in attempting to forecast the entire time-series data curve.

FIGURE 6.9: PPA proactive autoscaler latency for *compose-post-service*

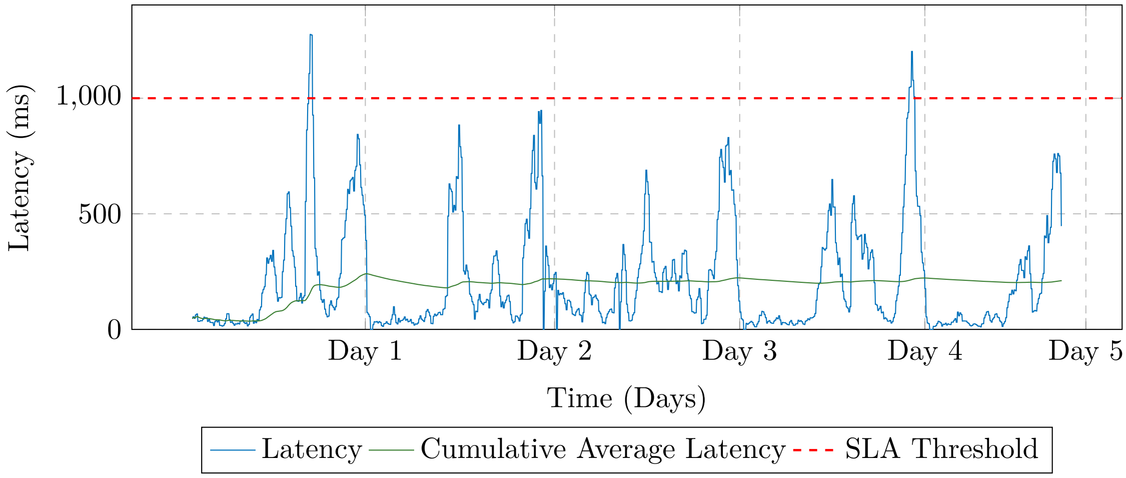


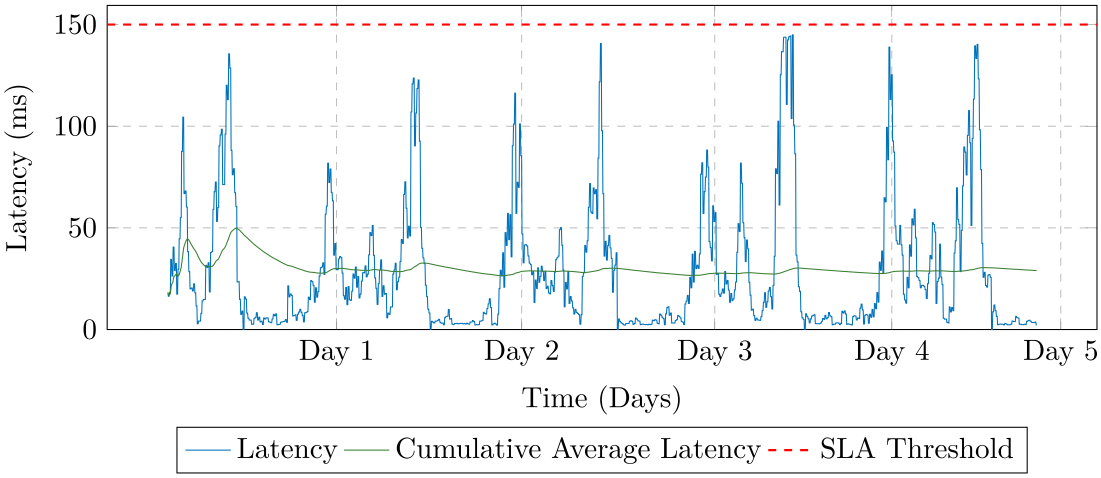
Figure 6.9 shows the full latency graph for the social network for the second experiment. As can be seen from the data, the latency initially spiked to above 1200 milliseconds, before stabilizing below the SLA threshold. However, one more spike occurred on the last day. Finally, the average latency of the autoscaler performed well, having a value of around 200 milliseconds which was similar to the reactive baseline.

Further investigations were conducted in an attempt to explain why these spikes occurred. It was deduced that the first latency spike occurred due to a shortage of training data. The LSTM used by the PPA architecture is extremely complex. Furthermore, apart from data normalization, no other data pre-processing was done such as passing it through the Savitsky-Golay smoothing algorithm. These issues make it more difficult for the forecaster to correctly predict data curves early on in its life cycle. This issue was resolved as more data was added to the training time-series input, however, after a certain point, a threshold was reached where the data was so large, it took a significantly large amount of time for the autoscaler to forecast the workload. This issue led to the latency spike on the final day leading to SLA violations, lasting for several minutes before the autoscaler corrected itself.

From these investigations, it can be seen that the proactive auto-scaling approach worked significantly better than the default Kubernetes autoscaler, but performed similarly to the reactive approach. While the social network POST requests were never dropped which would lead to complete system unavailability, the SLA constraints were nonetheless breached, due to the lack of edge layer resources and large training times for the proactive model. Due to issues such as this, the algorithm could not be deemed suitable for edge deployments that were reliant on SLA constraints.

6.6.4 Proposed Hybrid Autoscaler

FIGURE 6.10: Hybrid autoscaler latency for *home-timeline-service*



Finally, with the baselines being established in a default, reactive, and proactive approach, the hybrid algorithm was tested on the five days of generated workload. This approach demonstrated how it could mitigate the issues seen in both reactive and proactive auto-scaling approaches. The autoscaler was extremely lightweight, and easy to configure since there was no hyper-parameter tuning required. Building on top of this lightweight, the proactive forecaster was able to forecast the beginning of the workload spike accurately, thus leading to it eliminating the issue of a cold start. The forecaster could not accurately predict the middle and end of the daily workloads, however, this was not an issue, since the reactive algorithm was capable of taking over the auto-scaling process and maintaining the necessary resources to avoid SLA violations. Figure 6.10 shows the complete latency graph for the GET request experiment.

From the graph, it is clear that for the GET request experiment, no SLA violations were present for the duration of the five-day workload. Thus, in this case, the autoscaler controller did not intervene in the LSTM training procedure and modify the hyper-parameters of the forecaster. The average latency experienced by the social network was also comparatively low, achieving results similar to the THPA reactive implementation with the value hovering at around 30 milliseconds. This performance was a significant improvement over the baseline algorithms and demonstrated the efficacy of such a model in an edge architecture deployment.

Figure 6.11: Hybrid autoscaler latency for *compose-post-service*

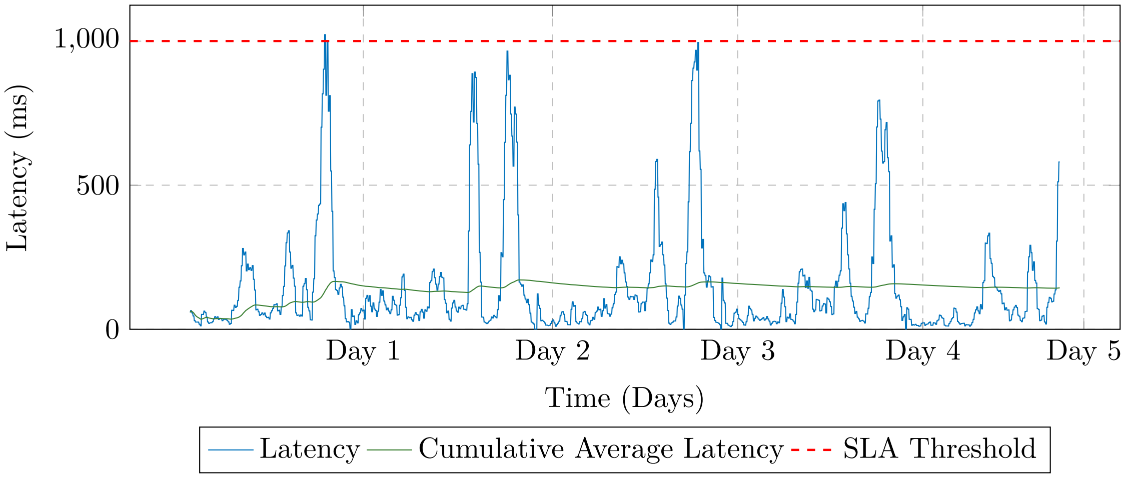


Figure 6.11 shows the latency metrics for the five-day workload simulation period for the second POST request experiment. From the graph, it can be clearly seen that only one SLA violation took place on the first day. This was a result of the lack of training data causing an erroneous prediction, an issue faced by the proactive baseline algorithm as well. However, what set this algorithm apart from that baseline was that the reactive autoscaler subsystem was capable of taking over from the proactive subsystem and scaling the resources accordingly. Due to this, the SLA threshold was only breached slightly, peaking at around 1020 milliseconds.

After the SLA violation, the autoscaler controller deduced that the training process needed to be kick-started through hyper-parameter tuning. This was done in the next training cycle when the controller provided the forecaster new hyper-parameter values, and as a result, it can be seen that no SLA violations took place on the next day. As a result, the autoscaler controller resets the hyper-parameters to speed up the training process, and even though the latency approached 990 milliseconds on the third day, no further violations took place for the rest of the simulation. The average latency of the whole experiment was also kept below 200 milliseconds, performing slightly better than its reactive and proactive baseline counterparts, and significantly better than the default Kubernetes approach.

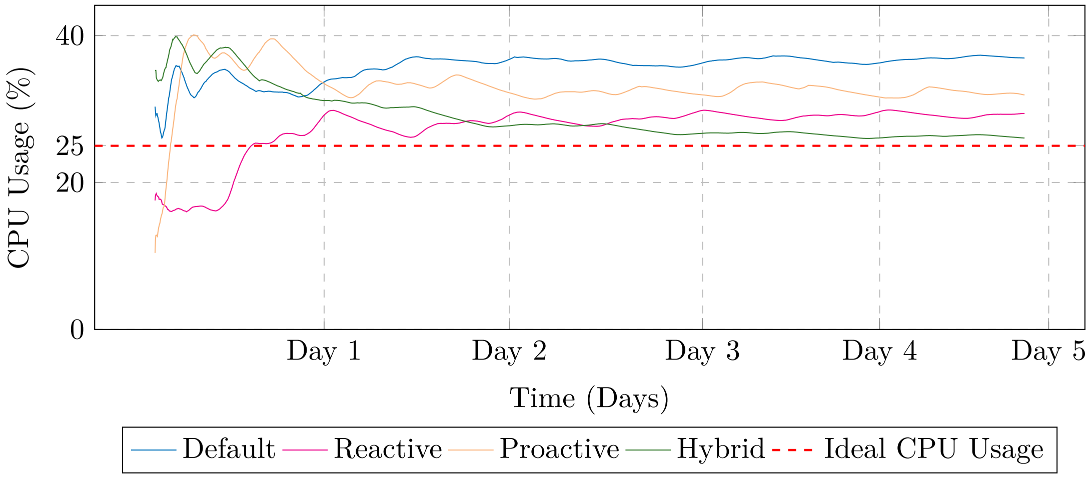
From these tests, it was clear that the hybrid approach had nearly eliminated the cold start problem very early on in the experiment. The initial difficulties in correctly forecasting the workload peaks were quickly resolved by the autoscaler controller's corrective instructions. All this was done with no user intervention, making the autoscaler extremely autonomous. Furthermore, the algorithm was capable of completing the training within a few minutes, allowing it to finish resource registration quickly. This meant that the CPU utilization of each pod on the nodes never exceeded 100%, and thus no user API requests were dropped, allowing for full system availability.

Based on these results, it was experimentally demonstrated that the proposed hybrid autoscaler performed far better than the baseline algorithms, breaching the SLA threshold in a negligible amount while causing no system unavailability. Thus, the suitability of this hybrid approach for an edge deployment under SLA constraints was proven.

6.7 Evaluation of CPU Workload Distribution

The distribution of CPU *workload* across the deployment pods was another important metric to factor into the analysis. As a reminder, the goal of the autoscaler was to maximize deployment resources while minimizing deployment costs. Considering the equal importance to be given to both SLA latency and optimization costs as explained in Section 4.1.1, and with an autoscaler threshold being set at $\mathcal{A}$, ideally, the distributed workload should hover at $\frac{\mathcal{A}}{2}$. When this *workload* distribution value approaches the value $\mathcal{A}$, it means that not enough pods are being deployed. This results in most if not all active pod resources in the deployment being utilized at full capacity, meaning new requests would need to be queued or even dropped, driving up latency. On the other hand, when *workload* tends towards the value 0, it means too many pods have been assigned to the deployment, and thus has been over-scaled. The returns when it comes to latency reduction for such a scaling scenario are extremely low, while the number of idle resource pods being active in the deployment is increasing significantly. This in turn increases the cost of running the underlying edge architecture, making this autoscaler an unprofitable alternative.

Figure 6.12: Autoscaler CPU workload distribution for *home-timeline-service*



For the first experiment, Figure 6.12 shows the average CPU utilization *workload* values of all the deployment pods for the three baseline algorithms and the proposed hybrid solution. As expected, the default Kubernetes horizontal pod autoscaler performs the worst, with the average utilization hovering around 35%. The proactive forecaster has utilization of approximately 33%, with it being held back by the forecaster's complexity and resource-intensive training. The reactive approach was the most lightweight autoscaler, and thus performed well with a CPU utilization of around 30%. Finally, the hybrid autoscaler performed the best of the four compared algorithms, with the utilization of approximately 26%.

In the second experiment with the heavier POST request, the auto-scaling threshold of the experiment $\mathcal{A}$ was set to 60%. Thus the ideal distributed workload achieved by the autoscaler should be half of the threshold, as calculated in the first experiment. Hence, the distributed workload for this experiment should be $\frac{60}{2} = 30\%$

Figure 6.13: Autoscaler CPU workload distribution for *compose-post-service*

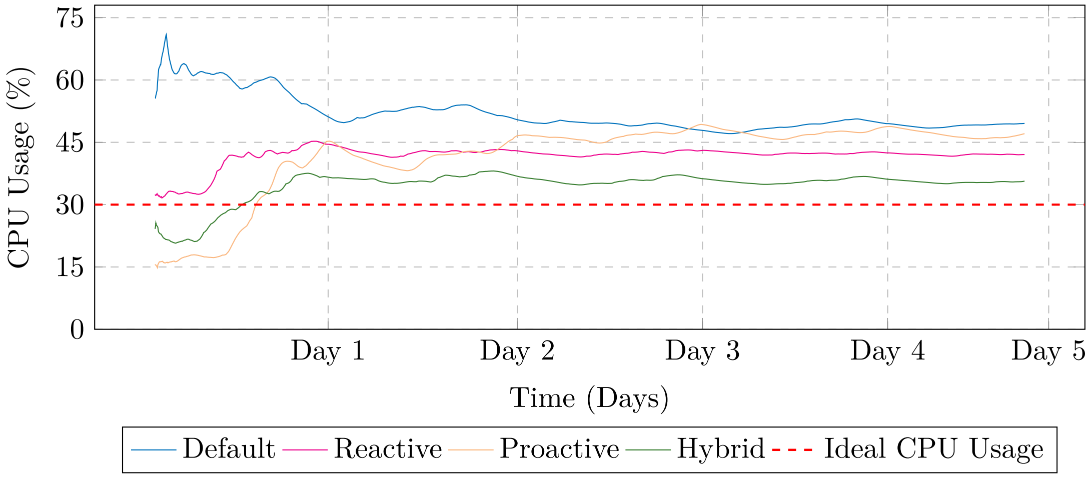


Figure 6.13 shows the average CPU utilization of the pods for the second experiment, being scaled in the deployment by the three baseline algorithms, along with the proposed hybrid solution. As shown above in Section 6.6.1, the default Kubernetes autoscaler faced issues with system unavailability, with user requests being dropped. This was caused by high CPU usage on the pods, with this being demonstrated by the graph. The average utilization peaked at around 70% before stabilizing at around 50% for all the resource pods. However, it is important to note that these resources were not distributed equitably, and thus while some pods had close to 100% utilization, others had near 0%.

The baseline proactive PPA algorithm performed better than the default Kubernetes autoscaler. While its CPU utilization stabilized fairly quickly to a value of approximately 45%, this was still far higher than the ideal distributed workload. This was most likely caused by the time it took for the algorithm to forecast data due to the complexity of the LSTM model, which resulted in resources lacking for certain periods of time.

The reactive THPA algorithm performed better than the proactive approach, with an average CPU utilization of approximately 42%. This was achieved due to the intelligent placement of pods in the edge nodes through its traffic-aware algorithm. However, the autoscaler was still prone to the cold start problem, and thus there were several occasions where the CPU workload would peak before the appropriate resources could be assigned. Due to this, the average workload was still far higher than the ideal value.

Finally, it can be seen from the graph that the hybrid autoscaler performed better than all three of the baseline algorithms. Its proactive subsystem allowed it to nearly eliminate the cold start problem, while its short training times, and reactive subsystem, allowed the efficient placement of resource pods on the deployment. The average CPU workload quickly stabilized to approximately 35%, which is far lower than the averages we have seen above for the baseline algorithms, and close to the ideal value of 30%.

Through this comparison, it was clearly demonstrated that for multiple use cases, the hybrid autoscaler managed to scale resources in a manner that both mitigated SLA violations, while doing so in a manner that was both light-weight enough for edge deployments, and inexpensive too.

6.8 Evaluation of SLA Violation Rates

As demonstrated above, the hybrid autoscaler performed significantly better than the baseline approaches. However, this was only compared using the flexible SLA thresholds. As a reminder, this was the most lenient threshold possible. For a more thorough demonstration, the algorithms needed to be tested on the other possible latency thresholds.

To achieve this, all four algorithms were tested again on the moderate and strict SLA violation thresholds for the GET request experiment, as displayed in Table 6.2. The IoT workload generation algorithm was once again used to generate this, however, this time the workload was run for five days, but auto-scaling was performed only for the last two days. This was done to get the best possible results for all autoscalers regardless of training data length. This resulted in the SLA violation percentage for approximately 1,020,000 GET requests.

Figure 6.14: SLA violation percentages for *home-timeline-service*

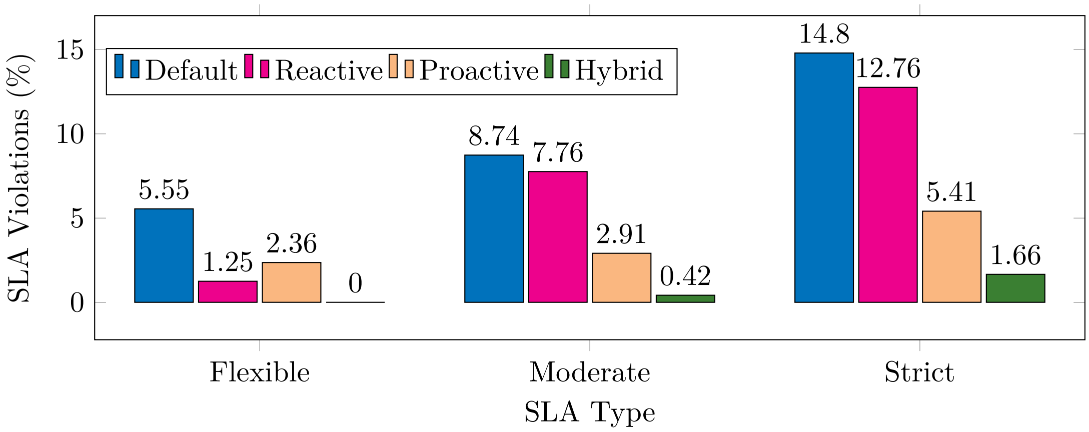


Figure 6.14 shows the SLA violations for the three different categories done on the GET request experiment. The flexible percentages were taken from the data taken from the five-day experiment using the 150 millisecond threshold, while the other two threshold values were extracted from the two-day experiments conducted for each. The default Kubernetes autoscaler performed the worst, with 5.55% of all requests being above the SLA threshold. The proactive PPA algorithm came next with a violation rate of 2.36%, with the number of violations being increased due to the complexity of the training model. The reactive THPA algorithm performed well, the low resource deployment, combined with the low overhead of GET requests ensured that it only violated the SLA thresholds of 1.25% of requests. Finally, as seen above, the hybrid autoscaler performed the best out of the four algorithms, being able to serve all the requests with a 0% SLA violation rate. This would qualify the hybrid architecture for a "highly available" SLA deployment.

While the flexible SLA threshold showed impressive results for three out of the four algorithms, the moderate threshold of 125 milliseconds was a more difficult constraint to maintain. The importance of mitigating or eliminating the cold start problem when it came to SLA latency was exposed in this threshold, as the default Kubernetes implementation and the reactive THPA autoscaler showed similarly poor results, violating 8.74% and 7.76% of requests respectively. On the contrary, the slower proactive PPA approach was able to demonstrate how mitigating the cold start problem while not being able to eliminate it completely could still have benefits, with the algorithm violating just 2.91% of requests. Finally, the hybrid autoscaler still achieved the best results. There were violations at the beginning of the experiment, resulting in 0.42% of requests being above the SLA threshold. However, this was quickly counteracted by the autoscaler controller, which modified the hyper-parameters to stabilize the latency. Once this was achieved, the hyper-parameters were dropped back down to the default values, and the auto-scaling continued as normal.

Finally, the strict SLA threshold of 100 milliseconds was tested, which proved extremely difficult to comply with. Once again, the default and reactive autoscalers performed significantly poorly, with 14.8% and 12.76% of requests not meeting the SLA threshold respectively. The proactive algorithm distinguished itself from the reactive ones, clearly showing the importance of the cold start problem, by only violating 5.41% of the requests. However, in this scenario as well, the hybrid algorithm performed substantially

Table 6.3: SLA violation counts for *home-timeline-service*

Request Counts	**Flexible**	**Moderate**	**Strict**
Total Requests	2,550,000	1,220,000	1,220,000
Default Violations	141,525	106,628	180,560
Reactive Violations	31,875	94,672	155,672
Proactive Violations	60,180	35,502	66,002
Hybrid Violations	0	5124	20,252

better with only 1.66% of violations. Even though the algorithm would occasionally violate the threshold, the combined approach, along with the controller's heuristic feedback was able to control the violation rate far better than the three baseline approaches. The total number of requests, along with the number of violations is provided in the Table 6.3.

Figure 6.15: SLA violation percentages for *compose-post-service*

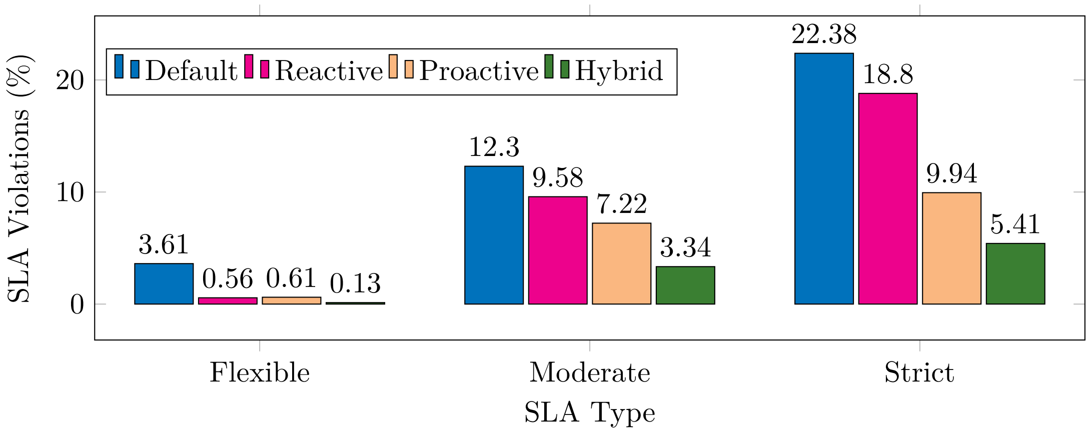


Figure 6.15 depicts the SLA violations for all four algorithms for the three SLA categories done on the POST request experiment. Once again, the flexible percentages were derived from the data taken during the five-day experiment using the 1000 millisecond threshold, while the moderate (900 milliseconds) and strict (800 milliseconds) violations were derived from the two-day workload above.

As seen above, for the flexible category, the default Kubernetes autoscaler performed the worst, with 3.61% of the total requests resulting in an SLA violation. This also includes the requests which resulted in an error due to being dropped by the social network. The reactive and proactive algorithms performed comparatively similarly to each other, achieving a 0.56% and 0.61% violation rate respectively. The lenient threshold means that the proactive algorithm is unable to display its benefits when it comes to

Table 6.4: SLA violation counts for *compose-post-service*

Request Counts	**Flexible**	**Moderate**	**Strict**
Total Requests	2,550,000	1,220,000	1,220,000
Default Violations	92,055	150,060	273,036
Reactive Violations	14,280	116,876	229,360
Proactive Violations	15,555	88,084	121,268
Hybrid Violations	3315	40,748	66,002

mitigating the cold start problem as compared to the reactive approach. Finally, the hybrid approach achieved a 0.13% SLA violation rate. This results in approximately 99.9% availability for the system, indicating that even for a difficult auto-scaling task such as for the POST request scenario, the hybrid algorithm is capable of achieving near "high availability".

The moderate SLA threshold proved far more difficult for all the algorithms to adhere to. Once again, the default Kubernetes autoscaler performed the worst of all four, failing to adhere to the SLA constraint for 12.3% of the requests. Here, the proactive autoscaler was able to demonstrate the importance of mitigating the issue of cold start. It was able to achieve a 7.22% of SLA violations, which was far lower than the 9.58% seen for the reactive autoscaler. However, once again, the hybrid solution proved to be the most capable of auto-scaling in this scenario, achieving an SLA violation rate of just 3.34%.

Finally, the extremely challenging strict threshold was tested. The default Kubernetes autoscaler was unable to cope with such a threshold, violating this threshold for over one-fourth of the requests, with a 22.38% violation rate. Once again, the proactive algorithm outperformed the reactive one, achieving a 9.94% violation rate as compared to the 18.8% of the reactive algorithm. Finally, the hybrid algorithm performed significantly better than all three baselines yet again, with a violation rate of just 5.41%.

This meant that over the two experiments and three SLA thresholds, the hybrid approach served a minimum of 94.5% of all requests in an SLA-compliant manner in the worst-case scenario while serving 100% of them in the best case. Through this thorough testing and experimentation, the algorithm displayed its robustness and adaptability while requiring little to no customization by the user to achieve excellent results. The total number of requests, along with the number of request violations for the second experiment is displayed in Table 6.4.

Thus for the two experiments, and on three different SLA thresholds, it can be comprehensively concluded that the hybrid approach proved to be the best autoscaler when it came to an edge deployment with minimal resources available. Furthermore, the algorithm was adaptable and able to display considerably improved performance in comparison with other reactive and proactive approaches. It was able to achieve this with minimal parameter configurations required by the user, and minimizing the cost of deploying and maintaining the edge architecture.

Chapter 7

Conclusions and Future Directions

7.1 Overview

In conclusion, the thesis provides a novel, lightweight, and SLA-compliant approach to autoscale resources on a micro-service deployed on an edge architecture. In comparison to other state-of-the-art autoscalers, the algorithm performs considerably better in terms of SLA violation as well as resource usage.

The autoscaler architecture is constructed using open source subsystems, and provides a framework to further develop more sophisticated auto-scaling solutions. These extensions will be discussed in Section 7.3.2.

Furthermore, the outcome of this thesis is being used to develop two publications. The first is a research paper titled “SLA-Driven Hybrid Autoscaling for Edge Computing: Foundations, Re- view, and Future Directions”, which is aimed to be published in the ACM Computing Survey [1]. The paper will focus on the cloud-edge layer deployment of Kubernetes, how the hybrid autoscaler fits into this architecture to provide SLA-compliance, as well as the mathematical modeling of the problem along with the performance evaluation.

[1] https://dl.acm.org/journal/csur

The second is an implementation framework publication titled “A Hybrid Reactive-Proactive Autoscaling Approach for Microservice Applications in Edge Computing Environments”, which is aimed to be published in the Journal of Systems and Software [2]. The autoscaler will be presented as an extendable framework on top of Kubernetes. The publication will discuss the modifications made to the autoscaler to make it SLA-compliant, and the methods by which users can develop their own auto-scaling policies using the hybrid autoscaler as a plug-in.

7.2 Contributions

The contributions of this thesis can be broken down into three main segments: identifying the major bottlenecks of auto-scaling on an edge deployment in comparison to a typical cloud architecture, designing a novel hybrid auto-scaling architecture which is built specifically for edge architecture paradigms, and streamlining the forecaster used by the hybrid autoscaler, thus making it capable of running on resource-limited edge deployments in a cost-effective manner.

Through the extensive literature review conducted in Chapter 3, the key aspects which make the typical cloud-based hybrid autoscalers infeasible for edge deployments were identified, this being a lack of computing and storage resources in the edge layer as compared to the cloud layer, the difficulty in hyper-parameter tuning, and the complexity of the forecast models driving up the model training time. With these issues identified, a hybrid auto-scaling architecture was proposed which addressed all the reported bottlenecks, and was capable of scaling resources in the edge environment in an SLA-compliant manner.

Based on the investigation of existing hybrid autoscalers conducted in Section 3.6, all existing autoscalers were not suitable for edge deployments, as they were too resource intensive, or not SLA-compliant, or both. Thus the hybrid architecture proposed in this thesis was a novel approach which maintained SLA compliance and ease-of-deployment on edge paradigms. It did so by combining a reactive autoscaler with a streamlined proactive model built using LSTM. To eliminate the difficult hyper-parameter tuning

[2] `https://www.journals.elsevier.com/journal-of-systems-and-software`

process, the autoscaler was built with a heuristic approach in mind, where the parameters were automatically fine-tuned in case any SLA violations were seen in the data.

Finally, the autoscaler was tested on a production-ready social network micro-service deployment, and the results were compared with other cutting-edge autoscalers. The results showed that the proposed autoscaler has a maximum SLA violation rate of 5.41%, as compared to the violation rate of $18.8 - 22.38\%$ for the other state-of-the-art autoscalers.

The tests were further able to demonstrate that the autoscaler was capable of significantly reducing SLA violations while also keeping the overall cost related to deploying the container orchestration resources on a cloud provider as low as possible. It does so by assigning resources in a manner which keeps the utilized resources at approximately half of the configured auto-scaling threshold.

Based on the work done in this thesis, the research questions stated in Section 1.2 can be conclusively answered.

- ***RQ1:*** The work done in this thesis successfully integrated reactive and proactive auto-scaling algorithms to develop a hybrid autoscaler tailored for edge computing architectures. The autoscaler was both light weight, allowing it to be run on the edge layer, while also eliminating the requirement for tedious and complicated hyper-parameter tuning seen in most proactive auto-scaling solutions.
- ***RQ2:*** The work done in this thesis successfully demonstrated that the autoscaler exceeded the SLA compliance capabilities of state-of-the-art reactive and proactive auto-scaling solutions for edge computing, while minimizing the cost of deploying the application.

7.3 Future Work

7.3.1 Current Limitations

Several assumptions have been made when conducting the experiments, the primary ones being that only a single SLA metric was used to design the heuristic model, the auto-scaling approach was limited to horizontally scaling resources, and the forecaster used a simple LSTM machine learning model using a single parameter. In this section, some extensions to improve the robustness of the hybrid autoscaler is proposed, along with some approaches to improve the efficiency and accuracy of its forecasts.

7.3.2 Proposed Extensions

7.3.2.1 Alternative Auto-scaling Approaches

The proposed hybrid model autoscales resources in a strictly horizontal manner. One of the underlying assumptions was that no other auto-scaling was to take place while the micro-service deployments were being stress tested. This meant that only the number of deployment replicas were either increased or decreased, keeping all other parameters untouched.

A future extension to this thesis would be to configure the algorithm such that it can attempt other forms of auto-scaling too. For example, the hybrid model may be configured as a Vertical Pod Autoscaler (VPA) instead. In this approach, the reactive autoscaler will modify the CPU reservations for the deployments, while leaving the number of deployment replicas untouched. Similarly, the proactive forecaster will attempt to predict the future CPU workload, and autoscale beforehand accordingly. The benefit of such an approach is that the forecaster already works in a similar manner, predicting future CPU workloads, and thus only minor modifications would be required to configure the hybrid HPA into a VPA.

A more challenging extension would be to transform the hybrid model into a Cluster Autoscaler. In the cluster autoscaler, the container orchestration platform will adjust the size of the cluster (edge nodes) when one of the following conditions holds:

- There are deployment replicas in the cluster which are currently un-assigned to any node due to insufficient resources.
- There are nodes in the cluster that have not had sufficient utilization of resources for a configured period of time, and as such, their active deployment replicas can be re-scheduled on to other nodes, after which the node can be decommissioned to free up resources.

While converting this autoscaler architecture for cloud computing is a straightforward process, due to the inherent homogeneity of the paradigm, it is much more difficult for an edge computing approach. Since different resource workloads are exerted on the various edge nodes, care needs to be taken when re-scheduling deployment replicas to other nodes, as well as decommissioning nodes. This process needs to be done in an intelligent manner such that the overall latency of the micro-service does not increase due to an added distance between the end user and the modified edge layer architecture. Therefore the auto-scaling controller must be modified in a way such that it can identify nodes nearby to it such that it can re-assign replicas.

7.3.2.2 Multi-Variate Forecaster

The forecaster used in the hybrid autoscaler is a uni-variate LSTM 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 multi-variate 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 multi-variate approach is that a more holistic auto-scaling decision can be made using this. For example, in the VPA algorithm discussed in Section 7.3.2.1, the reactive and proactive auto-scaling subsystems can be modified to autoscale on both CPU and memory utilization. The forecaster can then take the CPU and memory time-series data as input, and output the forecast for both. Through this, the autoscaler can form a better picture of the resource utilization of the micro-service, identifying bottlenecks and mitigating them.

The drawback of such an approach is that the proactive forecaster training times will significantly increase due to two reasons. The first one is the additional complexity of the input as well as output. Due to the multiple variables needing to be trained and validated on, more epochs are required, along with each epoch taking longer to evaluate. Furthermore the validation process will take longer as well. The second cause is due to the memory time-series not being as predictable as the CPU utilization. In the CPU utilization graphs of the deployments as seen in Figure 5.3, the workload forms clear and identifiable patterns which can be easily recognized by the model. Memory is far more variable, with sudden peaks and falls depending on several of the processes running in the deployment. Thus the autoscaler must concomitantly be made as complex, with far larger and deeper neural network architectures.

Due to this drawback, further research is required to verify the feasibility of using a multi-variate machine learning model to forecast workloads, and to confirm whether or not it is suitable for an edge deployment.

7.3.2.3 Multi-SLA Constraints

Another modification which can be made to the hybrid autoscaler is to support the use of multiple SLA constraints. Currently, the autoscaler only takes into account a single metric, and bases its heuristic feedback algorithm on it. By modifying this into a multi-parameter model, the autoscaler 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 hybrid 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.

Bibliography

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

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

[3] Mahadev Satyanarayanan. The emergence of edge computing. *Computer* , 50(1): 30–39, 2017.

[4] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access* , 8:85714–85728, 2020.

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

[6] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Computing Surveys (CSUR)* , 52(6):1–36, 2019.

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

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

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

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

[11] Paul Gruba and Justin Zobel. *How to write your first thesis.* Springer, 2017.

[12] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.

[13] G Justy Mirobi and L Arockiam. Service level agreement in cloud computing: An overview. In *2015 International Conference On Control, Instrumentation, Communication And Computational Technologies (ICCICCT)* , pages 753–758. IEEE, 2015.

[14] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information And Software Technology* , 131:106449, 2021.

[15] Mina Nabi, Maria Toeroe, and Ferhat Khendek. Availability in the cloud: State of the art. *Journal Of Network And Computer Applications* , 60:54–67, 2016.

[16] Emiliano Casalicchio. Container orchestration: A survey. *Systems Modeling: Methodologies And Tools* , pages 221–235, 2019.

[17] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. A kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software*, 175:110924, 2021.

[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications Of The ACM* , 59(5):50–57, 2016.

[19] Jonathan Baier. *Getting started with kubernetes.* Packt Publishing Ltd, 2017.

[20] Paridhika Kayal. Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope. In *2020 IEEE 6th World Forum On Internet Of Things (WF-IoT)* , pages 1–6. IEEE, 2020.

[21] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. Kosmos: Vertical and horizontal resource autoscaling for kubernetes. In *International Conference On Service-Oriented Computing* , pages 821–829. Springer, 2021.

[22] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. Adaptive microservice scaling for elastic applications. *IEEE Internet Of Things Journal* , 7(5):4195–4202, 2020.

[23] Amal Mahmoud and Ammar Mohammed. A survey on deep learning for time-series forecasting. *Machine Learning And Big Data Analytics Paradigms: Analysis, Applications And Challenges* , pages 365–392, 2021.

[24] Bendong Zhao, Huanzhang Lu, Shangfeng Chen, Junliang Liu, and Dongya Wu. Convolutional neural networks for time series classification. *Journal Of Systems Engineering And Electronics* , 28(1):162–169, 2017.

[25] Benjamin Lindemann, Timo Müller, Hannes Vietz, Nasser Jazdi, and Michael Weyrich. A survey on long short-term memory networks for time series prediction. *Procedia Cirp* , 99:650–655, 2021.

[26] Massimiliano Marcellino, James H Stock, and Mark W Watson. A comparison of direct and iterated multistep ar methods for forecasting macroeconomic time series. *Journal Of Econometrics* , 135(1-2):499–526, 2006.

[27] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE Access* , 6:6900–6919, 2017.

[28] Ruslan Smeliansky. Hierarchical edge computing. In *2018 International Scientific And Technical Conference Modern Computer Network Technologies (MoNeTeC)* , pages 1–11. IEEE, 2018.

[29] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference On Computer Communications* , pages 1–9. IEEE, 2016.

[30] Yaser Jararweh, Ahmad Doulat, Omar AlQudah, Ejaz Ahmed, Mahmoud Al-Ayyoub, and Elhadj Benkhelifa. The future of mobile cloud computing: integrating cloudlets and mobile edge computing. In *2016 23rd International Conference On Telecommunications (ICT)* , pages 1–5. IEEE, 2016.

[31] Guodong Wang, Yanxiao Zhao, Jun Huang, and Wei Wang. The controller placement problem in software defined networking: A survey. *IEEE Network* , 31(5): 21–27, 2017.

[32] Jiajia Liu, Shangwei Zhang, Nei Kato, Hirotaka Ujikawa, and Kenichi Suzuki. Device-to-device communications for enhancing quality of experience in software defined multi-tier lte-a networks. *IEEE Network* , 29(4):46–52, 2015.

[33] Ping Du and Akihiro Nakao. Application specific mobile edge computing through network softwarization. In *2016 5th IEEE International Conference On Cloud Networking (Cloudnet)* , pages 130–135. IEEE, 2016.

[34] Yaser Jararweh, Ahmad Doulat, Ala Darabseh, Mohammad Alsmirat, Mahmoud Al-Ayyoub, and Elhadj Benkhelifa. Sdmec: Software defined system for mobile edge computing. In *2016 IEEE International Conference On Cloud Engineering Workshop (IC2EW)* , pages 88–93. IEEE, 2016.

[35] Ola Salman, Imad Elhajj, Ayman Kayssi, and Ali Chehab. Edge computing enabling the internet of things. In *2015 IEEE 2nd World Forum On Internet Of Things (WF-IoT)* , pages 603–608. IEEE, 2015.

[36] Gopika Premsankar, Mario Di Francesco, and Tarik Taleb. Edge computing for the internet of things: A case study. *IEEE Internet Of Things Journal* , 5(2):1275–1284, 2018.

[37] Tejaswini Mishra, Meng Wang, Ahmed A Metwally, Gireesh K Bogu, Andrew W Brooks, Amir Bahmani, Arash Alavi, Alessandra Celli, Emily Higgs, Orit Dagan-Rosenfeld, et al. Early detection of covid-19 using a smartwatch. *MedRxiv* , pages 2020–07, 2020.

[38] Endah Kristiani, Chao-Tung Yang, Yuan Ting Wang, and Chin-Yin Huang. Implementation of an edge computing architecture using openstack and kubernetes. In Kuinam J. Kim and Nakhoon Baek, editors, *Information Science and Applications 2018*, pages 675–685, Singapore, 2019. Springer Singapore. ISBN 978-981-13-1056-0.

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

[40] Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, and Jordy Dieltjens. Reducing cold starts during elastic scaling of containers in kubernetes. In *Proceedings Of The 36th Annual ACM Symposium On Applied Computing* , pages 60–68, 2021.

[41] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Rapid serverless deployment using environment snapshots. *ArXiv Preprint ArXiv:1910.01558* , 2019.

[42] C Lorenzo, P Guillaume, and P Bellavista. Fogdocker: Start container now fetch image later. In *Proceedings Of The 12th IEEE/ACM International Conference On Utility And Cloud Computing (UCC)* , 2019.

[43] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *ArXiv Preprint ArXiv:1903.12221* , 2019.

[44] Salvatore Venticinque, Rocco Aversa, Beniamino Di Martino, Massimilano Rak, and Dana Petcu. 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* , pages 587–594. Springer, 2011.

[45] Sherif Sakr and Anna Liu. Sla-based and consumer-centric dynamic provisioning for cloud databases. In *2012 IEEE Fifth International Conference On Cloud Computing* , pages 360–367. IEEE, 2012.

[46] Ryan Houlihan, Xiaojiang Du, Chiu C Tan, Jie Wu, and Mohsen Guizani. Auditing cloud service level agreement on vm cpu speed. In *2014 IEEE International Conference On Communications (ICC)* , pages 799–803. IEEE, 2014.

[47] Linlin Wu, Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments. *IEEE Transactions On Services Computing* , 7(3):465–485, 2013.

[48] Rajkumar Rajavel and T Mala. 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* , pages 547–554. Springer, 2012.

[49] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Resource provisioning for iot services in the fog. In *2016 IEEE 9th International Conference On Service-oriented Computing And Applications (SOCA)* , pages 32–39. IEEE, 2016.

[50] Mohammad Aazam and Eui-Nam Huh. Dynamic resource provisioning through fog micro datacenter. In *2015 IEEE International Conference On Pervasive Computing And Communication Workshops (PerCom Workshops)* , pages 105–110. IEEE, 2015.

[51] Lina Ni, Jinquan Zhang, Changjun Jiang, Chungang Yan, and Kan Yu. Resource allocation strategy in fog computing based on priced timed petri nets. *IEEE Internet Of Things Journal* , 4(5):1216–1228, 2017.

[52] Nguyen Dinh Nguyen, Linh-An Phan, Dae-Heon Park, Sehan Kim, and Taehong Kim. Elasticfog: Elastic resource provisioning in container-based fog computing. *IEEE Access* , 8:183879–183890, 2020.

[53] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021-IEEE Conference On Computer Communications* , pages 1–9. IEEE, 2021.

[54] Jānis Kampars and Krišjānis Pinka. Auto-scaling and adjustment platform for cloud-based systems. In *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings Of The International Scientific And Practical Conference* , volume 2, pages 52–57, 2017.

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

[56] Satish Narayana Srirama, Mainak Adhikari, and Souvik Paul. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal Of Network And Computer Applications* , 160:102629, 2020.

[57] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a contemporary deployment platform using docker containers. In *Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings 13* , pages 316–323. Springer, 2015.

[58] Guilherme Santos, Hervé Paulino, and Tomé Vardasca. 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* , pages 242–249, 2020.

[59] Gerta Sheganaku, Stefan Schulte, Philipp Waibel, and Ingo Weber. Cost-efficient auto-scaling of container-based elastic processes. *Future Generation Computer Systems* , 138:296–312, 2023.

[60] Salman Taherizadeh and Vlado Stankovski. Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal* , 62(2):174–197, 2019.

[61] Joao Paulo KS Nunes, Thiago Bianchi, Anderson Y Iwasaki, and Elisa Yumi Nakagawa. State of the art on microservices autoscaling: An overview. *Anais Do XLVIII Seminário Integrado De Software E Hardware* , pages 30–38, 2021.

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

[63] Li Ju, Prashant Singh, and Salman Toor. Proactive autoscaling for edge computing systems with kubernetes. In *Proceedings Of The 14th IEEE/ACM International Conference On Utility And Cloud Computing Companion* , pages 1–8, 2021.

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

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

[66] Valter Rogério Messias, Julio Cezar Estrella, Ricardo Ehlers, Marcos José Santana, Regina Carlucci Santana, and Stephan Reiff-Marganiec. Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. *Neural Computing And Applications* , 27:2383–2406, 2016.

[67] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions On Services Computing* , 15(3):1448–1460, 2020.

[68] Yoosef Alidoost Alanagh, Mojtaba Firouzi, Abdolreza Rasouli Kenari, and Mahboubeh Shamsi. Introducing an adaptive model for auto-scaling cloud computing based on workload classification. *Concurrency And Computation: Practice And Experience* , 35(22):e7720, 2023.

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

[70] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference On Machine Learning* , pages 1310–1318. Pmlr, 2013.

[71] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[72] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm-a literature review. In *2019 International Conference On Machine Learning, Big Data, Cloud And Parallel Computing (COMITCon)* , pages 380–384. IEEE, 2019.

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

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

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

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

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

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

[79] Parminder Singh, Pooja Gupta, and Kiran Jyoti. Tasm: technocrat arima and svr model for workload prediction of web applications in cloud. *Cluster Computing* , 22(2):619–633, 2019.

[80] Salam Hamdan, Moussa Ayyash, and Sufyan Almajali. Edge-computing architectures for internet of things applications: A survey. *Sensors* , 20(22):6441, 2020.

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

[82] Walayat Hussain, Farookh Khadeer Hussain, and Omar Khadeer Hussain. 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* , pages 309–316. Springer, 2016.

[83] Anjali Patel and Nisha Chaurasia. 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) , pages 39–44. IEEE, 2021.

[84] Silvano Martello and Paolo Toth. Algorithms for knapsack problems. *North-Holland Mathematics Studies* , 132:213–257, 1987.

[85] Hans Kellerer, Ulrich Pferschy, David Pisinger, Hans Kellerer, Ulrich Pferschy, and David Pisinger. Introduction to np-completeness of knapsack problems. *Knapsack Problems* , pages 483–493, 2004.

[86] EG Radhika and G Sudha Sadasivam. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. *Materials Today: Proceedings* , 45:2793–2800, 2021.

[87] José F Torres, Dalil Hadjout, Abderrazak Sebaa, Francisco Martínez-Álvarez, and Alicia Troncoso. Deep learning for time series forecasting: a survey. *Big Data* , 9 (1):3–21, 2021.

[88] Christopher Boucher. Curve fitting of solution data in comsol multiphysics - comsol.com. 2024. [Accessed 04-05-2024].

[89] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. A comparison of arima and lstm in forecasting time series. In *2018 17th IEEE International Conference On Machine Learning And Applications (ICMLA)* , pages 1394–1401. IEEE, 2018.

[90] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 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* , pages 3–18, 2019.

[91] Uma Tadakamalla and Daniel A Menascé. 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* , pages 1–15. Springer, 2019.

[92] Mor Sides, Anat Bremler-Barr, and Elisha Rosensweig. Yo-yo attack: vulnerability in auto-scaling mechanism. In *Proceedings Of The 2015 ACM Conference On Special Interest Group On Data Communication* , pages 103–104, 2015.

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

[94] Ronald W Schafer. What is a savitzky-golay filter?[lecture notes]. *IEEE Signal Processing Magazine* , 28(4):111–117, 2011.

[95] P Kingma Diederik. Adam: A method for stochastic optimization. *(No Title)* , 2014.

[96] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE Transactions On Neural Networks And Learning Systems* , 28(10):2222–2232, 2016.

[97] Jose Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards network-aware resource provisioning in kubernetes for fog computing applications. In *2019 IEEE Conference On Network Softwarization (NetSoft)* , pages 351–359. IEEE, 2019.

[98] Oscar Nilsson and Noel Yngwe. Api latency and user experience: What aspects impact latency and what are the implications for company performance?, 2022.

[99] Christof Lutteroth and Gerald Weber. Database synchronization as a service. In *2009 13th Enterprise Distributed Object Computing Conference Workshops* , pages 84–91. IEEE, 2009.