Master Thesis

Dynamically scaling Azure resources using Machine Learning

Thijs Klumpenaar

a thesis submitted in partial fulfilment of the requirements for the degree of

Master Business Analytics
(Optimization of Business Processes)

Vrije Universiteit Amsterdam

Faculty of Science

Supervised by: Dr. Rianne de Heide, Emiel Stoelinga (Info Support)
2^{nd} reader: Dr. Joost Berkhout

Submitted: July 25, 2024

Abstract

This research addresses the challenge of efficiently scaling cloud resources in real-time to minimize cost and energy consumption. We used Recurrent Neural Networks and Long Short-Term Memory models to forecast the computational load of cloud resources, which proved to be quite successful. We developed a scaling model that maps the computational load forecasts to a capacity level, or service tier, for the cloud resource. The proposed solution yielded significant improvements in resource utilization and achieved cost savings of up to 80% compared to current static approaches. Although the model effectively predicted computational load and scaled resources efficiently, it suffered from excessive scaling, which may significantly affect resource usability in practice. Further research is required to refine the proposed solution and mitigate this issue.

Preface

This thesis is written during a six-month internship at Info Support. This thesis is my final document created for the master's degree of Business Analytics at the Vrije Universiteit. While I have greatly enjoyed the journey and challenges of this curriculum, all good things must come to an end.

I would like to express my gratitude towards my supervisors Emiel Stoelinga and Dr. Rianne de Heide for guiding me throughout my research and being available for consult whenever necessary.

Furthermore, I would like to thank all other members of Info Support for all the insightful and enjoyable discussions, which greatly contributed to the development of this research.

Lastly, I would like to thank my friends, family and Bram for their constant support and positive distractions throughout the process.

List of Figures

Contents

Chapter 1

Introduction

The global cloud computing market has seen rapid growth, with its size estimated at approximately 630 billion dollars in 2023. The market is expected to expand further, reaching approximately 3 trillion dollars by 2033 [1]. As demand for cloud computing continues to increase, this expansion also drives a significant rise in the industry's carbon footprint. Nowadays, the carbon footprint of cloud computing exceeds that of the airline industry [31].

Customers can leverage cloud computing services through cloud providers like Azure, Amazon Web Services and Google cloud. Users can use all sorts of services and computational power, such as virtual machines, databases, micro-services and networking technologies, for which they are charged based on how much they reserve or use. In practice, the customer's focus is usually on ensuring there is enough capacity reserved to meet the demand, while overcapacity and its associated costs are often overlooked. This leads to higher energy consumption and costs than necessary, which is financially inefficient and environmentally damaging.

This study is significant because it addresses the issue of excessive energy use and financial waste due to over-reservation of computational resources in cloud environments. By optimizing the reservation of cloud resources, we can achieve substantial cost savings and reduce energy consumption.

The research question that was formulated for this research, is:

> "To what extent can advanced analytics, such as Machine Learning, be used to predict cloud computation load in real-time and dynamically scale cloud resources to minimize cost and energy usage?"

This research will focus specifically on Azure Web Apps, DTU-based Azure SQL Databases, and vCore-based Azure SQL Databases, as for these resources data was available. The findings and conclusions of this research aim to improve the efficiency of resource reservation in cloud environments and provide valuable insights for industry practices.

In Chapter 2, we provide a background of relevant information for this research. In Chapter 3, we examine the available data and analyze the different features that are present in this data. In Chapter 4, we outline the choices made during the modelling phase of this research and present the modelling results. In Chapter 5, we interpret the results from the modelling Chapter and assess the implications of these results. In Chapter 6, we summarize the findings of this research and answer the research question.

In Chapter 7, we explore limitations and future work, and offer our recommendations to Info Support based on this study.

Chapter 2

Background

In this chapter, a background of relevant information for this research is given. In Sections 2.1 and 2.2 we introduce Cloud computing in Azure and the Azure resources that will be focussed on in this research. In Section 2.3 we explain the different ways of scaling the computational power of the resources. In Section 2.4 and 2.5 we discuss different types of models that can be used to forecast time series and methods to scale resources based on these forecasts. In Section 2.6 we consider different ways of splitting the available data into train, validation and test sets.

2.1 Cloud computing in Azure

Cloud computing is the delivery of computing services that are not hosted on-premise, such as databases, servers, web applications, networking and Artificial Intelligence. Cloud computing offers many benefits opposed to on-premise resources, such as cost efficiency, flexibility (easy up- and downscaling of resources), scalability and security [5, 34]

The three most prominent players in global cloud computing are Amazon Webservices, Microsoft Azure and Google Cloud, having a global market share of 34%, 21% and 11% respectively, totalling to a 66% combined market share [15]. The global cloud infrastructure spending was approximately 57 billion dollars in 2022.

The cloud resources that will be focussed on in this research are hosted in Azure, which is Microsoft's public cloud platform. In the next section, the different types of the available Azure resources will be discussed.

2.2 Cloud resources

In this research, we consider data from three different types of cloud resources, namely Azure Web Apps, DTU-based Azure SQL Databases and vCore-based Azure SQL Databases. As Azure Web Apps have an underlying Virtual Machine that they run on, we address Virtual Machines as well. The computational units, which are measures of processing power such as the number of CPU cores, are explained per resource type. The up- and downscaling of the resources by adding or removing computational units is discussed in the next section.

2.2.1 Virtual Machines

An Azure Virtual Machine (VM) is a computer-like machine that is hosted in Azure [19]. Just like a normal computer, a VM has RAM, a CPU, storage and can also connect to the internet if required. An important difference between VMs and normal computers, is that normal computers are made up from hardware, such as RAM sticks or a physical CPU, while VMs exist only as code, as software-defined computers within physical servers. The main computational units that make up a Virtual Machine in Azure, are virtual centralized processing units (vCPUs) and random access memory (RAM).

2.2.2 Azure Web Apps

Azure Web Apps are web-based applications that are hosted in the Azure App Service (AAS), which is a fully-managed platform as a service (PaaS) for building applications in Azure [26]. AAS supports a wide range of programming languages, such as .Net, Java, PHP and Python. Furthermore, users pay only for the computational resources they use, which are determined by the App Service Plan in which the app is run.

The App Service Plan can be seen as one or more underlying Virtual Machines that run the Web Apps. Therefore, to change the computational resources that is available to the Web Apps, is to change the computational resources or count of the Virtual Machines underlying to the App Service Plan. Therefore, the computational units that are relevant for Azure Web Apps, are also vCPUs and RAM.

2.2.3 Azure SQL Databases

SQL Databases are computational systems that store and organize structures sets of data in collections of tables, that use SQL Server [20]. Naturally, an Azure SQL Database is a SQL Database that is hosted in Azure, providing many benefits, such as automatic patching, maintenance, and scalability. All SQL Databases communicate using structured query language (SQL).

There are two different cost models for Azure SQL Databases, namely DTU-based and vCore-based models [21]. A Database Transaction Unit (DTU) is a computational unit that encapsulates a combination of CPU power, memory and data I/O resources. Increasing the number of DTUs increases the capacity to handle concurrent users and process query workloads. DTUs are designed to simplify the process of selecting an adequate performance level for the database, which is convenient for this research. The vCore based model is similar to the model that VMs use, as a vCore is practically a part of a vCPU. The amount of RAM that is available per vCore is set at 5.1GB.

2.2.4 Costs

In Cloud environments, users pay for the amount of computational resources or services they use. This means that when a user for example needs a more powerful database, the corresponding costs of reserving this will be higher. However, a resource instance should be able to successfully execute its task using the reserved computational power. Thus, selecting an adequate computational tier is a trade-off between the resource instance's ability to perform its task and corresponding costs. If a database experiences fluctuating computational demand the service tier can be changed, which is called either upscaling or downscaling, depending on whether the computational power is increased

or decreased respectively. The different types of scaling are discussed in more depth in the next section.

2.3 Scaling

In Azure, there are typically two types of scaling a resource, namely horizontal and vertical scaling [22]. Horizontal scaling means adding or removing instances of resources used by a workload. An example of this is extending the number of VMs that run an application from two to three VMs. In contrast to horizontal scaling, vertical scaling keeps the number of resource instances constant, but changes the number of computational resources that is available to the resource instance, such as CPU power or RAM. An example of this is changing the amount of RAM for a VM from 8GB to 16 GB. Per resource type, there are predefined service tiers, which correspond to a fixed number of available computational resources, such as the amount of RAM, CPU power and storage.

Both horizontal and vertical scaling have distinct benefits which can make one option more suitable over the other, depending on the tasks of the resource. Horizontal scaling offers a higher fault tolerance, as other compute instances can potentially take over the tasks of a failing instance. However, increasing the number of compute instances also increases the number of for example Windows or Linux systems that are running, causing computational overhead. Scaling vertically does not have this issue, as the number of separate instances that are running remains the same.

2.3.1 Virtual Machines

Virtual Machines can generally be scaled vertically, and in some cases also horizontally. A standalone Virtual Machine can be scaled vertically by changing the number of vCPUs and amount of RAM that is available to the VM. There are multiple specialisations among Virtual Machines, such as 'Compute optimized', 'Memory optimized' and 'General Purpose' [24]. The type of VM that is best suited will depend on task for which the VM was created.

VMs can also be part of a 'Virtual Machine Scale Set' (VMSS), which is a collection of VM instances. Horizontal scaling can be done by increasing or decreasing the number of instances that run an application. By adding or removing VM instances from the scale set, the total computational capacity of the VMSS can be increased or decreased respectively [23].

2.3.2 Web App

The resources available to the Web App can be scaled up by increasing the computational resources of the App Service Plan of which the Web App is part of [27]. The App Service Plan can be scaled either horizontally or vertically. Horizontal scaling can be performed by scaling the number of Virtual Machines that run the Web App. The maximum number of Virtual Machines that can run the Web App is 30. Vertical scaling is performed by changing the number of vCPUs and amount of RAM.

For Web Apps, three different types of computational plans are available, namely Basic, Premium and Isolated. The Basic Service Plan has 3 tiers, ranging from $0.018 to $0.07 per hour. When higher reliability, horizontal scaling and better performance is important, the Premium Plan can be used. The 9 tiers of the Premium Plan range from

$0.210 to $5.824 per hour. Thus, running the highest Premium tier on full capacity costs approximately $140 per day. The most expensive plan is the Isolated Plan, ranging from $0.562 to $17.984 per hour. This plan offers very high scalability in an isolated and dedicated environment, which can be necessary in specific scenarios.

The type of plan for which applying the scaling model will be most relevant, is the Premium Plan. The costs corresponding to the higher tiers in this plan make it potentially profitable to invest time and effort into dynamically scaling the Plan. Furthermore, the Premium Plan is generally used significantly more than the Isolated Plan, as in many cases the benefits of the Isolated Plan that lead to higher cost are excessive. However, all methods described in this research are also applicable to other scaling plans.

2.3.3 Azure SQL Databases

As discussed earlier, two different computational models are available for Azure SQL Databases, namely DTU-based and vCore-bases databases. For vCore-based databases, the number of virtual cores, amount of memory, amount of storage and storage speed can be adjusted. DTU-based databases can be scaled by changing the number of DTUs and changing the amount of storage that is available to the database.

2.3.4 Current autoscaling methods

For Virtual Machines, Azure has an inbuilt horizontal autoscaling option. This autoscaling can be performed either rule-based or schedule based. Rule-based scaling occurs based on user-defined thresholds for selected computational metrics. Users can for example create a rule that scales the number of Virtual Machines in the scale set when the average CPU utilization is above 70% for more than 10 minutes. Schedule-based scaling is scaling that occurs on a user-defined schedule, such as every day at 7:00. There are however drawbacks to both available autoscaling methods.

According to Microsoft [25], performance-based rules may impact application performance before the autoscale rules trigger and new Virtual Machine instances are provisioned. Furthermore, when sudden high load occurs, the Virtual Machine must first be potentially underprovisioned for the threshold duration of the scaling rule, whereafter the Virtual Machine pool must scale, potentially further impacting performance. The schedule-based autoscaling is a less flexible method that is based on static assumptions of future load metrics, not taking into account the current and future computational metrics. This potentially causes the scale set to scale too early, or during high computational load. Dynamically predicting and anticipating on computational load can potentially increase the likelihood that the Virtual Machines are not underprovisioned, and scaling at appropriate moments.

Predicting Autoscaling is a currently available feature for Virtual Machine scale sets and App Service Plans, which can dynamically predict the overall CPU load and horizontally scale the scale set, anticipating on predicted demand. However, the CPU load is the only variable that is used for the forecasts, both as predicting and predicted variable. This means that other computational load metrics that are relevant for Virtual Machines, such as RAM usage, are not considered. Furthermore, this feature can only be used to horizontally scale a VM scale set, making it a feature only to be used in one specific scenario for VMs.

2.4 Forecasting models

This sections discusses models than can potentially be used to predict the computational load of cloud resources and applications of these models in literature.

2.4.1 ARMA and ARIMA

Autoregressive Moving Average (ARMA) models are regressive models that model a time series based on the idea that the current value of a time series can be explained by past values of the same time series [36]. An extension of this model, the Autoregressive Integrated Moving Average (ARIMA), first differences the relevant time series a number of times before fitting an ARMA model on the differenced time series. In both cases, the time series is modelled by only its own values and corresponding errors.

Calheiros et al. [4] used an Autoregressive Integrated Moving Average (ARIMA) model to predict computational loads of a collection of Virtual Machines in a cloud environment. The Mean Absolute Percentage Error (MAPE) was equal to 9%. However, the forecasted metric showed strong cyclic behaviour, together with more inaccurate predictions whenever the metric deviated slightly from the cycles. This means that the obtained results are less generalizable than the results of other papers.

Roy et al. [32] obtained very accurate results predicting the number of clients in a cloud environment over a period of 500 hours using an Autoregressive Moving Average (ARMA) model. In this case, there was also a clear cyclic pattern present in the forecasted values. However, deviations from this cyclic pattern were fitted correctly. Furthermore, a cost model was used to allocate numbers of virtual machines over time, based on the predicted number of clients. In this model, costs were associated with violating Quality-of-Service (QoS) measures.

The limitation of ARMA and ARIMA models is that they predict a single time series only based on patterns in the time series itself, such as moving averages of historical values. When there are meaningful feature values that are believed to have predictive power, such as the time of day, other models can be used. Relevant examples of those models are discussed in the next section.

2.4.2 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a neural network architecture mainly used for detecting patterns in sequential data [33]. The difference between Feedforward Neural Networks and Recurrent Neural Networks is how information is passed through the network. While Neural Networks have different layers that pass through numeric information, a RNN transmits the output of a layer back into itself, hence the term "recurrent". This allows the model to detect and learn patterns over multiple observations in a sequence.

The following hyperparameters are relevant for RNNs:

- Number of layers
- Number of units per layer
- Dropout rate
- Learning rate
- Activation functions
- Optimizer

The number of layers refers to the number of LSTM layers that are stacked on top of eachother, while the number of units per layer indicates how many nodes are present in a single RNN layer. Increasing the number of layers or units per layer allows the model to learn more complex patterns present in the data, but also increases the computational cost and the risk of overfitting.

Dropout is a regularisation technique to improve the generalization of neural networks [6, 17], aiming to avoid overfitting in neural networks. Dropout refers to the act of randomly multiplying the output of nodes with 0, or "dropping" the output of these nodes. Implementing this for the training process causes the model to learn the output based on a dynamic selection of inputs instead of all inputs, providing a more generalizable predictive model.

The step size in which the model updates its weights and biases during training is given by the learning rate. Choosing a learning rate too low can result in slow training and convergence to local optima. Choosing a learning rate too high can cause the learning process to not converge, or jump over well performing optima in the highly-dimensional loss space.

The activation function is the function that is used to transform the weighted inputs of a node, which can be chosen per RNN layer. Widely used activation functions are the Hyperbolic tangent (tanh), sigmoid function and the Rectified Linear Unit (ReLU) [13]. The goal of activation functions is to introduce non-linear behaviour in the output of the RNN layers.

The algorithm that adjusts the parameters (weights and biases) of neural networks is called the optimizer. Simpler optimizers, such as stochastic gradient descent, uses the gradient of the loss function to adjust the weights in a way that the value of the loss function should improve. More advanced optimizers, such as Adapative Momement Estimation (Adam) or Nesterov Adam (Nadam), smartly use for example moving averages of gradients to determine adaptive learning rates to improve the learning process.

Zhang et al. [41] used an RNN to predict the computational workload in a Google cloud cluster, by using the CPU and RAM metrics over time. They showed that the RNN based method was suitable for forecasting the load, yielding accurate results and consistently outperforming an ARIMA model. The lowest Mean Squared Error that was obtained for the standardised CPU and RAM were equal to $2.76 * 10^{-5}$ and $1.51 * 10^{-5}$ respectively. The proposed future work for this research was trying more complex recurrent models, such as the Long Short-Term Memory, which will be discussed in the next section.

Duggan et al. [8] predicted host CPU utilization in a cloud environment using RNNs with relatively high accuracy results when predicting 15 minutes into the future. The forecasting error and error variability increased linearly when increasing the length of the forecasting window, which is to be expected.

An important issue that can arise when training RNNs, is the vanishing or exploding gradient problem Hochreiter [11]. The vanishing or exploding gradient problem occurs when consecutive gradients are propogated throughout many layers of the network. As the gradients in the network are multiplied during the backpropogation, the final gradients can exponentially decline towards zero (vanishing) or grow to a very large number (exploding). A vanished gradient causes the eventual weights to be updated only marginally, strongly influencing the rate of convergence and eventual model performance. An exploding gradient causes the eventual weights to be updated excessively, leading to unstable training and potentially resulting in diverging model parameters.

To properly handle these potential problems, specialized RNN architectures were employed, such as the Long Short-Term Memory.

2.4.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) models are a widely used extension of the Recurrent Neural Network, designed to properly handle the vanishing or exploding gradient problem [10, 35]. Another important improvement is the introduction of "gates", which control the information flow for example by determining which internal signal to amplify or weaken, or to determine what signals output gets sent to the next unit. Also, the memory mechanism of the LSTM is more sophisticated. Furthermore, the model is able to handle noise over time as well [12]. All mentioned improvements allow the model to learn long-term dependencies more effectively, generally translating to better performance in practice compared to that of RNNs. As the architectural changes did not introduce any new hyperparameters, the RNN hyperparameters apply to the LSTM as well.

Yadav et al. [40] predicted the hourly average load of a distributed server in a cloud environment for an interval of 24 hours using LSTM models. The models were quite performant, obtaining a mean absolute error of 0.043 for the normalized load.

Nguyen et al. [30] used an LSTM with an encoder-decoder model to forecast the Google cluster workload traces with satisfying results. The dataset was very diverse, containing over 25 million tasks across more than 12.000 hosts over May 2011. The time interval between records was five minutes. The proposed method yielded consistently better results over a conventional LSTM model, especially for longer prediction windows. An important thing to note is that the results were only marginally better for prediction windows between approximately 40 and 160 minutes, or 8 and 32 observations respectively. The forecasting window in this research will be smaller than the lower bound, as will be discussed in Section 3.5.2.

Zhong et al. [42] implemented LSTM models to forecast the number of requests that arrived at a VM. The forecasted number of requests was then used as input for a Q-Learning agent, that determined whether to scale or not to scale the VM. The proposed approach decreased SLA violations by up to 20% to 30%, indicating a sufficiently performant forecast.

2.4.4 Prediction window

The time interval for which the computational load should be predicted, or the prediction window, is dependent on the time it takes for the resource instance to scale. The prediction window should start after the time it takes to scale the resource instance, or the scaling window, as is is not useful to forecast the computational load for moments for which we cannot scale the resource instance early enough. For example, predicting the computational load for the next 5 minutes, while the resource instance takes 10 minutes to scale is not useful, as we cannot scale the resource early enough to match the predicted demand. Thus, in this case, the prediction window should start after 10 minutes from the current moment. Next, we will determine the length of the scaling window per resource type.

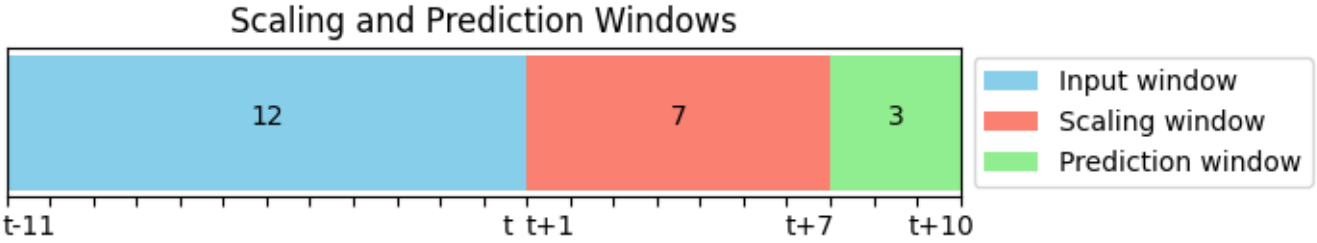

Figure 2.1: Input, scaling and prediction window for the Web Apps

Startup times Virtual Machines

Because a new VM compute instance must be created when a VM is scaled either horizontally or vertically, the average startup time will be considered as the scaling time for a Virtual Machine. Furthermore, because the App Service Plan for a Web App has an underlying Virtual Machine, we also consider the startup times for a Virtual Machine as the scaling time for an App Service Plan.

Mao and Humphrey [18] found that the average startup time for Azure VMs was approximately between six and seven minutes. Because the frequency in which the available data is present is one minute, the forecasting window must be at least seven for the Virtual Machines and Web Apps.

Azure SQL Database

For Azure SQL Databases, scaling from one computational tier to another generally takes under five minutes per database, or under one minute per GB of space used, depending on the current and objective computational tiers [28].

Having determined the length of the scaling window, we can now investigate the length of the prediction window. The load values predicted in the prediction window are the values on which the scaling decision will be based. To prevent the scaling model from directly up- and downscaling when a single predicted load value is high or low, we can base the scaling decision on the average value over multiple observations instead. This way, the scaling mechanism is more robust, as volatility in the forecasts has a less direct and drastic impact on the scaling decisions. The length of the prediction window would then be a measure of the sensitivity of the scaling model. When it is vital that the predicted computational load is satisfied, a short prediction window is necessary. Thus, it can also be thought of as how important it is that the forecasted computational load is satisfied.

Figure 2.1 gives a visual representation of the input window, scaling window and prediction window for the Web Apps. The specific lengths of the input and output windows will be discussed in sections 3.5.2 and 3.5.3.

2.5 Scaling model

2.5.1 Rule-based

A rule-based scaling model indicates whether a resource should be scaled based on thresholds for computational metrics. Lower and upper thresholds are defined for

which the resource should scale up or down if the computational load exceeds or drops down the threshold respectively. Choosing the correct threshold values for both the lower and upper bound is important, as these values can directly influence if or when an upscaling or downscaling action is triggered, thus affecting performance and cost objectives [38]. Using a high upper bound can reduce total cost, while potentially also decreasing performance, and vice versa.

Mohan Murthy et al. [29] defined such thresholds, namely that the VMs in question should be downscaled when the vCPU or RAM utilization dropped down 25%, or upscaled if it exceeded 80%. Suleiman and Venugopal [38] formulated two thresholds for scaling a web server based on the CPU load. Both lower bounds were equal to 30%. The upper bound was chosen to be either 75% or 80%. However, these methods only account for scaling up or down a single step in the service tier hierarchy, per scaling decision.

2.5.2 Tier based

Another method is to determine what future service tier would be most appropriate for the resource based on the predicted computational load, and then scale to that specific tier. This way, we can span multiple tiers in a single scaling decision, without needing to incrementally scale up or down.

When scaling decisions are based on multiple computational load metrics, we can select the cheapest suitable tier that satisfies the predicted computational load per metric and union them. We can then take the highest tier of this union, to make sure that we satisfy all future computational load metrics. For example, when the forecasts of three computational load metrics are used for the scaling decision, we take the three cheapest service tiers that satisfy each individual metric and union them. We then take the highest service tier within this union as our final tier that satisfies the computational load for all metrics. The assumption here is that the amount of computational resources is non-decreasing when choosing a higher, more expensive service tier, which is the case for all service tiers in this research. In this study, we will consider a tier-based scaling model.

2.5.3 Scaling performance measures

To evaluate the performance of the scaling model, we formulate multiple numerical performance measures. These performance measures can be used to evaluate the effectiveness of the scaling mechanism and compare the obtained performance to the current situation. The performance measures and their pseudo-definition are given in Table 2.1. When applicable, these performance measures will be calculated for each target feature.

The utilization of the resources can be seen as the efficiency of the total cost. Since the necessary amount of computational power is fixed, a higher resource utilization indicates a lower average capacity, leading to greater cost savings. The lacking capacity percentage when the resource was underprovisioned represents how much capacity was missing on average for all observations for which the available capacity was insufficient. The importance of this performance metric is application dependent.

Measure	Pseudo-definition
% of time underprovisioned	$\frac{\text{Time underprovisioned}}{\text{Total time}}$
% lacking when underprovisioned	MAPE for all observations where lacking
% utilization of resources	$\frac{\text{Capacity used}}{\min(\text{Total capacity, Capacity used})}$
% of time spent scaling	$\frac{\text{Time spent scaling}}{\text{Total time}}$

Table 2.1: Amount of computational units per service tier, with a capacity scaling factor of 0.7

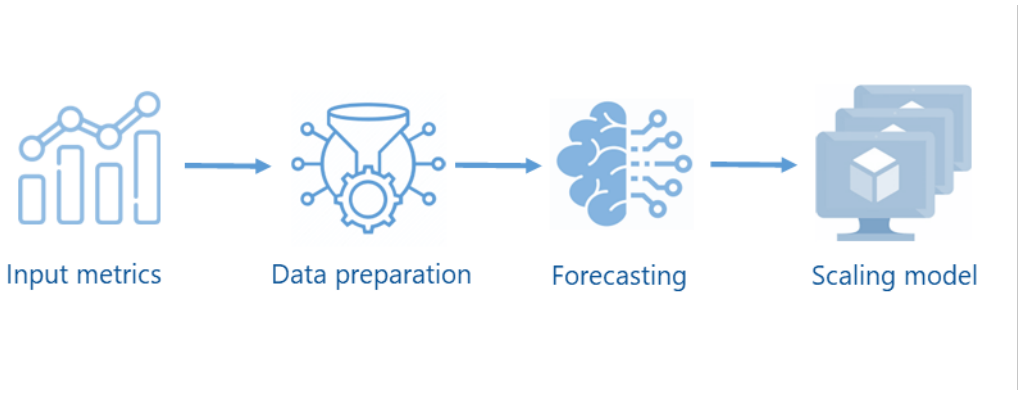

Figure 2.2: Full overview of dynamic scaling mechanism

2.5.4 Full mechanism overview

The decision to up- or downscale is made each minute in real time. Each minute, when another observation of all computational metrics is available, we use the most recent observations as input for the forecasting model. This model then forecasts the target features during the forecasting window, as shown in Figure 2.1. The scaling model will then map the average computational load over the forecasting window to the corresponding service tier, as described in the previous section. When the expected suitable service tier differs from the current service tier, we scale to the target service tier. When the decision to scale is made, we will wait until the resource is scaled, until we continue the process. Figure 2.2 gives an overview of the full dynamic scaling mechanism.

In Section 7.3, we will further delve into possible architectures for the scaling mechanism, using available Azure services.

2.6 Dataset splitting

While modelling data using Machine or Deep Learning techniques, it is important to preprocess and split the data so that the investigated models can be trained, validated and tested correctly. It is important that the data that is used to finally estimate the models true performance, also known as the test set, is kept aside during the model

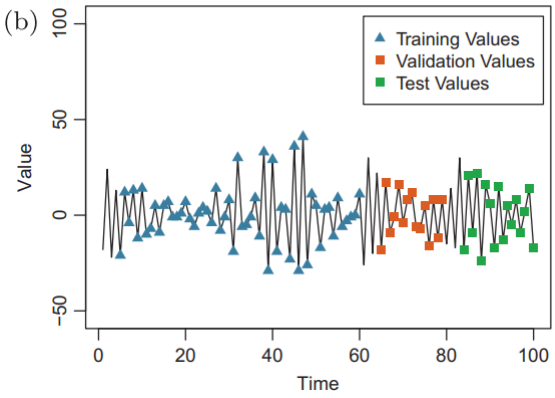

Figure 2.3: Dataset splitting for Last-block evaluation [3]

building and selection procedure. This prevents pollution of the final evaluation results, as the models should be evaluated on new data, meaning data that the model has not seen yet.

A common way of splitting time series data into train, validation and test sets is last-block evaluation [3]. Last-block evaluation splits the time-series dataset into training, validation and test sets in a chronological order. This way, the model will be evaluated or tested on data that comes after the data the model was trained on. Figure 2.3 gives a visual representation of a time-series, split into the different sets using last-block evaluation.

Evaluating the model during training based on how well it predicts future values is logical, as this precisely is its task in production. During training, the model is evaluated based on its performance to predict the next values, namely the validation set. When model architectures are selected using the training and validation set, the final performance is estimated based on unseen future values, the test set. The final results should therefore be representative for the eventual performance in production.

Furthermore, splitting the data in this manner prevents individual observations to be used in more than one of the mentioned sets. Using observations in multiple sets causes the model to be evaluated on observations it has already seen, polluting the reliability of the performance metrics. In between the sets in Figure 2.3, a small gap of unlabeled observations can be seen. This is done so that the observations of the time series per set are non-overlapping.

Snijders [37] experimented with the use of blocked cross-validation, which is a more advanced version of last-block evaluation. In his study, there were no clear result improvements over the last-block evaluation. As the last-block evaluation is a less complex and more straightforward approach, this was the recommended technique for validating time series models during the model selection procedure.

In this research, we will be using the last-block evaluation method as shown in the figure. The training, validation and test set sizes will be 60%, 20% and 20% of the total dataset respectively, as these are widely used proportions.

2.7 Conclusion

This research will focus on dynamically predicting computational load for Azure Web Apps and Azure SQL Servers, and scaling the resources based on these predictions. The forecasting models that will be considered during this research, are the Recurrent Neural Network and Long Short-Term Memory models, as these models are consistently performant and popular in similar research that is present in literature. Scaling the cloud resource instances will be done using a tier-based scaling model, based on forecasts of the relevant computational load metrics.

The research discussed in this thesis differentiates itself from similar work described in this chapter by the level in which the forecasting and scaling models are applied and the type of scaling model. The described research generally uses Machine Learning to predict a single time series that represents the total computational load in a cluster of resources in a cloud environment. Furthermore, similar research utilised a rule-based scaling model. This research will focus on predicting the computational load of multiple individual cloud resource instances using ML models and a tier-based scaling model.

Chapter 3

Data preparation

In this chapter, we explore the the available data per resource type. In Section 3.1 we list all features present per resource type. In Section 3.2 we outline the steps taken to prepare the data for the modelling part of this research. In Section 3.3 we visualise important variables and relations between variables. In Section 3.4 we explore other datasets originating from Web Apps hosted in a production environment. In Section 3.5 we discuss the form of the input and output data.

3.1 Data availability

3.1.1 Web App

The variables that were logged for the Web App are as follows:

- Timestamp on which the record was logged
- Average response time of the Web App
- Total CPU time that the Web App used
- Total Data used as Input for the Web App
- Total Data used as Output for the Web App
- Average handle Count of the Web App [1]
- Total Bytes that the Web App read during runtime
- Total Bytes that the Web App wrote during runtime
- Average amount of memory the Web App occupied
- Total number of requests that the Web App processed

In the previous chapter we discussed that the App Service Plan's scalable computational resources were the number of vCPUs and the amount of RAM. These computational resources directly correspond to the 'total CPU time' and the 'Average memory working set' variables that are present in the data.

The first six Web apps run in the same App Service Plan. This App service plan is hosted in the same cloud environment as all of Azure SQL databases. Web app 7 is hosted in a separate App Service Plan, in another cloud environment.

[1] Handles are a concept in operating systems, which generally represents a reference to an operating system resource, such as files, memory objects, network connections and database connections. Poorly optimized code or infrastructure design can introduce a large amount of simultaneous handles, causing potential latency. However, nowadays this generally does not cause problems anymore, due to the large handle capacities of storage devices, and better optimized code and infrastructure.

3.1.2 Azure SQL Database (DTU based)

The following variables were logged for the DTU-based Azure SQL databases:

- Time at which the metrics were recorded.
- CPU utilization percentage
- Percentage of DTUs used
- Number of DTUs used
- Percentage of log write operations
- Count of active sessions or connections
- CPU utilization percentage for the SQL Server instance
- Memory utilization percentage for the SQL Server instance
- Percentage of CPU core utilization by the SQL Server process
- Percentage of system memory utilized by the SQL Server process
- Size of the temporary database's data files
- Size of the temporary database's transaction log files

For the DTU-based databases, the feature that directly correspond to the database performance is the Number of DTUs used.

3.1.3 Azure SQL Database (vCore based)

For the vCore-based Azure SQL databases, the following variables were present:

- Current CPU usage
- CPU usage percentage
- Maximum allowed CPU usage
- Billed CPU usage by application
- Application CPU usage percentage
- Application memory usage percentage
- Data storage space
- Storage usage percentage
- Reserved data storage space
- Percentage of time spent writing logs
- Storage usage percentage for Extreme Transaction Processing
- Percentage of time spent reading physical data
- Percentage of utilized workers
- Number of active sessions
- Percentage of used sessions
- Indication of successful connection
- Size of log backups in bytes
- Size of full backups in bytes
- Size of differential backups in bytes

The variables that directly correspond to the amount of consumed computational units are the CPU usage and the memory usage percentage.

3.1.4 Logging duration

The moment from which the data was present, differed per resource. Table 3.1 shows per resource the number of consecutive days for which data was present. The end of the logging duration was the first of April for all resource instances.

Resource	Number of days
Web app 1	20
Web app 2	20
Web app 3	34
Web app 4	34
Web app 5	34
Web app 6	34
Web app 7	60
AzureSQL DTU 1	20
AzureSQL DTU 2	20
AzureSQL DTU 3	34
AzureSQL vCore 1	20
AzureSQL vCore 2	20

Table 3.1: Number of consecutive days for which data was present per resource

3.1.5 Target features

The target features that will be predicted during this research, are the features that directly correspond to the usage of the computational units.

Table 3.2 gives these target variables per resource type.

Resource type	Predicted variables
Web Apps	CPU time & RAM usage
AzureSQL database (vCore)	vCPU time & RAM usage
AzureSQL database (DTU)	DTUs used

Table 3.2: Predicted variables per resource type

Section 3.3 will focus on exploring these predicted variables and their relation to the input features.

3.2 Data preprocessing

Kotsiantis et al. [14] identified the following six steps of data preprocessing for supervised machine learning tasks:

- Outlier detection
- Missing feature values
- Discretization [2]
- Feature selection [3]
- Feature engineering
- Normalization

[2] As we did not find any valuable applications for discretization for this data, we leave this part out.

[3] As all available data could potentially possess predictive power and the number of available features was not high, we did not drop any features.

As it became apparent that the data regarding the Azure SQL databases was not usable for this research, we focus on the Web app data in this section. We will further discuss the Azure SQL database data in Section 3.3.

3.2.1 Outlier detection

Outliers that were present in the data were not removed, as they can contain valuable information about the computational load for the resources. As we will see in Section 3.3, there are relatively very high values present in for example the CPU time of the Web apps. These values should however not be removed, as it is important that high values of the predicted variables are modelled correctly.

3.2.2 Missing feature values

A small number of observations around the start of the logging period that contained missing feature values were removed. For many resource instances, the first hour contained some ranges of missing values. To combat this, the first observations that contained a relatively high amount of missing feature values were removed. This was determined by manually investigating the first observations per resource instance. When there were first observations to be dropped, the logging period that was dropped was between one and two hours long.

For Azure Web Apps, there were three features that contained a higher amount of missing values than other features, namely "Data In", "Data Out" and "Number of requests". When investigating the data, it was observed that the variables only contained missing feature values when the other features implied inactive behaviour. During periods where for example many requests were arriving to the Web App, all feature values were consistently and correctly registered. Because of this behaviour, the missing feature values were regarded as inactive behaviour. Thus, for a missing value the value that corresponded to inactive behaviour was imputed. For example, the number 0 was imputed for missing values of the number of requests. Apart from at the start of the logging period, no longer periods of missing values were found. However, some individual missing values were found that were linearly interpolated, so that the missing values would be imputed with realistic values.

The reason that we impute the missing values instead of dropping them, is that all time series elements need to be consecutively logged values. When dropping observations, the formed time series do not consist of directly consecutive values anymore, potentially causing the eventual forecasting model to learn patterns that in reality are not there.

3.2.3 Feature engineering

As Section 3.3 shows that there is a profound difference between the computational load for the Web apps during the weekend and weekdays, we expand the input features with information about the day of the week and time of day on which an observation was logged. The five features that were added, were the hour of the day, together with four boolean features that indicated whether the observation was logged during the morning, afternoon, evening or night. Furthermore, all features that were logged in bytes, such as Data In/Out, were transformed to either Kilobytes or Megabytes, dividing them by 1024 or 1024^2 respectively.

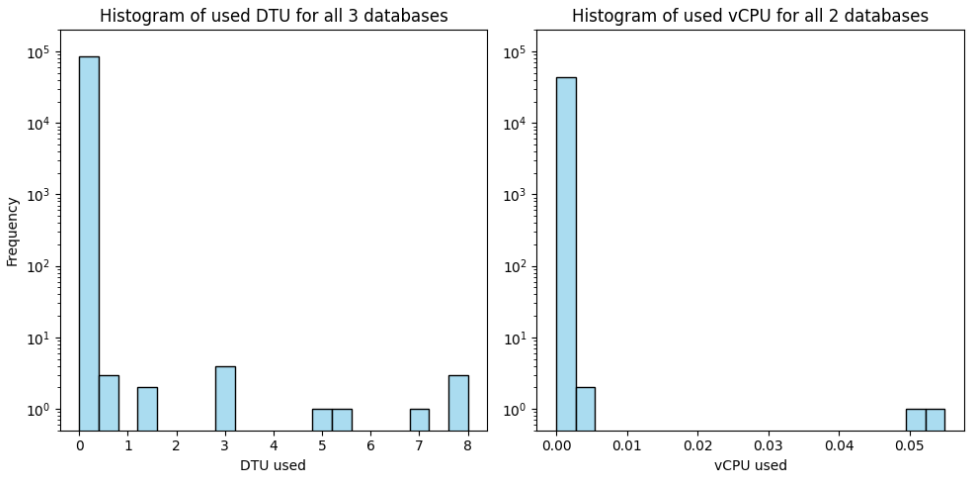

Figure 3.1: Azure SQL database: Histogram of the used DTU and vCPU for all databases

3.2.4 Normalization

To scale the different datasets to a range that is more suitable for the forecasting models, the features of all datasets were normalized using the mean and standard deviation of the training set. The data was normalized by first subtracting the mean and then dividing by the standard deviation. This way, there is no data leakage between the training, validation and test datasets, which is the usage of information during the training process which would not be available at the moment of prediction. Using either validation or test data for scaling the training data would be similar to using data from the future for scaling the data in production, which is impossible.

3.3 Exploratory Data Analysis

In this section, we explore the data by visualising the available features and relations between them. We aim to uncover potential patterns and important relations, which can provide substantiation for the choices made in the later chapters. Furthermore, gaining insight into the data used for training the models can help us interpret the results more effectively.

3.3.1 Azure SQL databases

When inspecting the target variables for the Azure SQL database types, we do not see any active behaviour in the computational usage. Figure 3.1 shows a histogram of the DTU and vCore usage of the Azure SQL databases. An important thing to note in this figure is the logarithmic y-scale.
The lack of activity in the computational metrics of the Azure SQL Databases is an indication that the databases are not utilized as much as the Web Apps. Needless to say, the only valuable comment that can be made regarding the Azure SQL databases, is that they should most likely be scaled to the lowest computational tier possible. The Web apps do however exhibit activity, which will be investigated further in the next section. In some cases, methods like oversampling can be used to combat feature balance

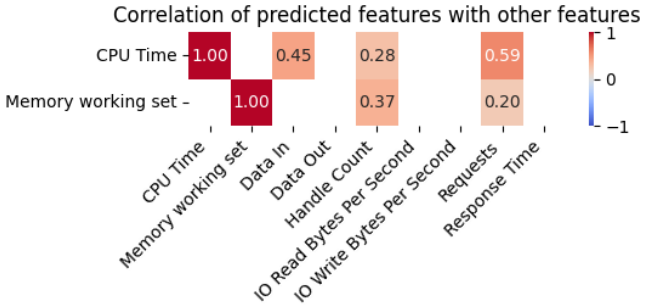

Figure 3.2: Web App: Correlation Matrix of target features with non-target features

present in datasets. However, the feature imbalance for the Azure SQL Databases is too severe to be able to obtain meaningful results when using the oversampled observations in both the training and validation sets.

Due to the exceptionally strong feature imbalance in both the input and target features, the available data from the Azure SQL database cannot be used to obtain valuable results. A forecasting model can only be reliably used for making predictions using data that is similar to the data that was used to train the model. Thus, a model trained on data from an inactive cloud environment cannot be used to fulfill the goal of this research, namely dynamically scaling the resource based on fluctuating computational demand. This is why we will not further use the data from the Azure SQL databases.

3.3.2 Web App

Figure 3.2 shows the correlations between the target and input features with an absolute value larger than 0.2. With this plot we aim to identify which features have a meaningful relationship with the target variable, which can be beneficial for the eventual performance of the forecasting model.

The two strongest correlations are between the CPU time and both the amount of ingested data and number of requests. The correlation between the number of requests and CPU time seems to be logical, as the number of requests determines how many times the Web app is used, thus influencing the CPU time. Also, the amount of data that is ingested also influences the total CPU time, as the ingested data must be processed and can also be used in further calculations. One would perhaps also expect a correlation to exist between the exported data and the CPU time, but this depends on the purpose of the Web app. For example, if no data is exported, it is most likely that no correlation will be present.

It is also logical that the handle count, which is the number of for example database connections or memory storage references, has a positive correlation with the RAM usage. As retrieved data must be stored in memory, having a higher amount of database connections likely increases the RAM usage.

Seeing no strong linear correlations between the predicted features and input features is not necessarily problematic, as there can be many other complex and non-linear relations in and between the different features. As discussed in the previous chapter, the modelling techniques used in this research are capable of capturing similar patterns.

Figure 3.3 visualises the distribution of the CPU time and RAM usage for all Web

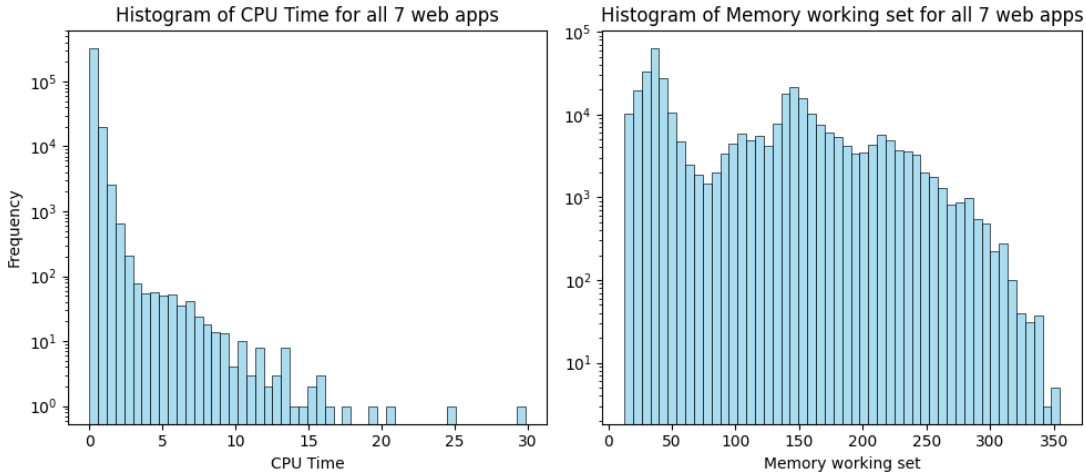

Figure 3.3: Web App: Histogram of the CPU Time and RAM usage for all Web apps

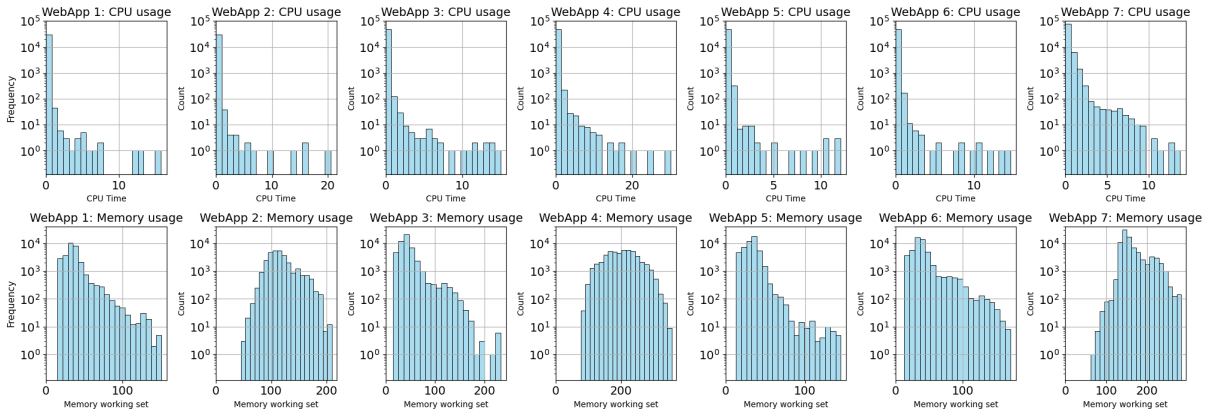

Figure 3.4: Web App: Histogram for the CPU time and used memory per Web app

apps combined. This helps us understand the overall scale of the distributions and identify any potential outliers.

The CPU time of the combination of all Web Apps resembles a decaying distribution with some high outliers. Despite the high number of observations with a relatively low CPU time, only 299 observations had a CPU time equal to 0.

Figure 3.4 shows the distributions of the CPU time and memory usage for all Web apps. Understanding the differences between the distributions per Web App can help explain potential differences in the forecasting results.

As can be seen in Figure 3.4, the Web apps roughly resemble the same decaying distribution for the CPU time, meaning that all Web apps generally have low CPU activity, while some higher values are present in increasingly lower quantity. Web app 7 deviates the most from this pattern, having a distribution that contains more mid-valued observations, relative to the other Web apps.

For the RAM usage some Web apps, like Web app 1, 3 and 6 approximately follow an exponentially decaying distribution, since it is decreasing linearly on an logarithmic scale. Other Web apps, like Web app 2 and 4 seem to be a distributed like a tailless normal distribution. No Web apps have values around 0 MB RAM usage, as a Web

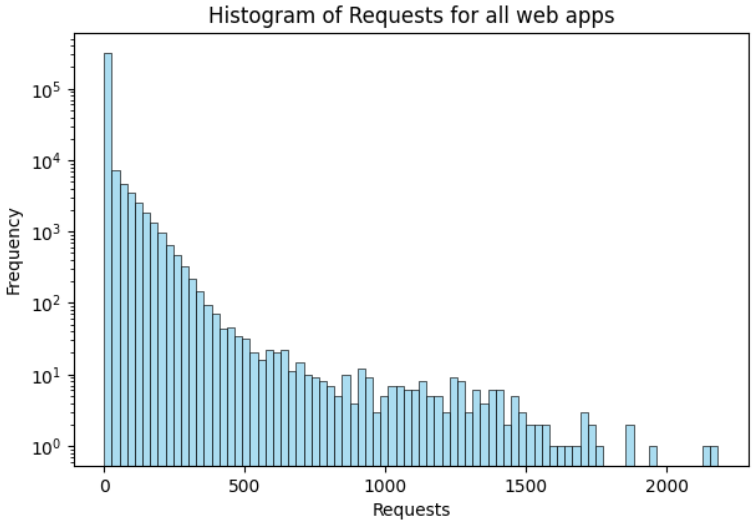

Figure 3.5: Web App: Histogram for the number of requests for all Web apps combined

app requires a certain minimal non-zero amount of RAM to be able to run.

To make sure that the activity in the target variables is not caused by for example periodic garbage collection or keep-awake processes, we also investigate the number of requests that arrive at the Web apps. Figure 3.5 shows a histogram of the number of requests for all Web apps combined.

We can see that while the highest frequency occurs at the lowest values, there are still plenty of observations with a higher number of requests. The distribution of the number of requests resembles a quite steadily decreasing distribution. The number of observations with a number of requests equal to 0 was approximately 176.000, while there were approximately 167.000 observations with a non-zero number of requests. This means that the activity that was visible in Figure 3.4 is most likely not only caused by other processes, but also by executions of the Web apps.

To gain insight into potential temporal patterns that are present in the computational behaviour of the Web Apps, we plot the CPU time over time. Figure 3.6 visualises the CPU usage per Web app over the period of an arbitrary weekday.

While some Web apps, like Web app 1 and 2, do not show any clear patterns in the CPU activity over time, there seem to be some patterns visible for the other Web apps. For example, Web app 3, 4 and 6 show increased activity starting from approximately 08:00, which would be the start of the working day. Web app 4 does not show this same increase, but does have some peaks starting from 08:00 until approximately 16:00. The most clear patterns can be seen in Web app 7 that shows no significant activity before 08:00, has non-zero CPU time over the entire working day, after which the CPU time drops to very low values again.

To further investigate potential temporal patterns for Web App 7, we visualise the usage of the Web App over time. Figure 3.7 shows the number of incoming requests for Web app 7 in the period of April 4^{th} to April 25^{th}. We chose to visualise this period as the relevant patterns were clearly visible over this period.

In this figure, we can see that there is a distinct difference between the behaviour of the number of requests during the weekdays and during weekends. This behaviour is logical, as Web app 7 is a Web app that is utilized by users throughout the working

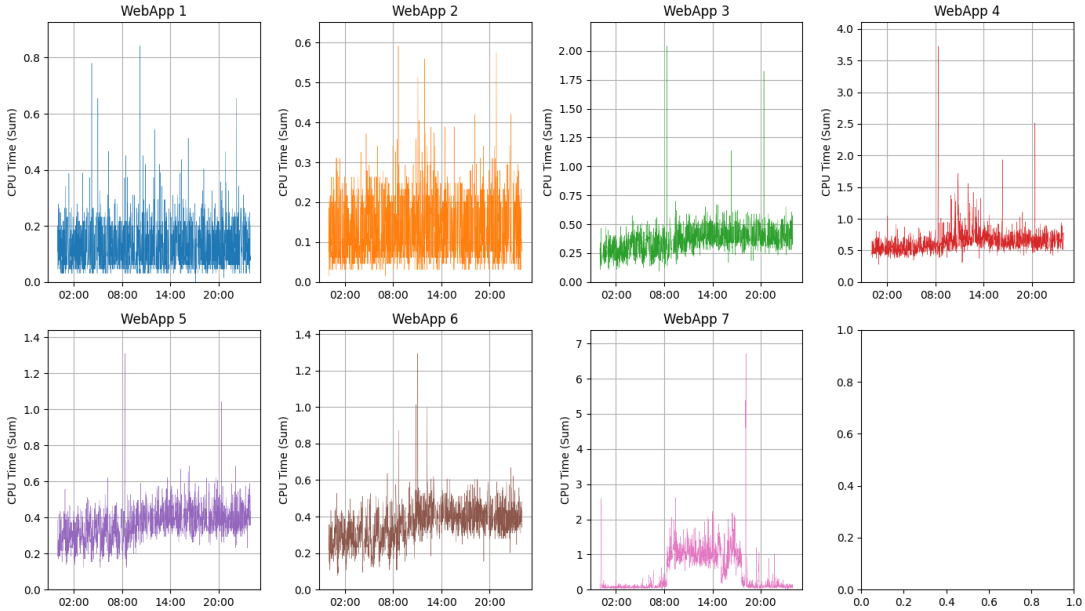

Figure 3.6: Web App: Line Plot of CPU Time for All Web Apps (1 Day)

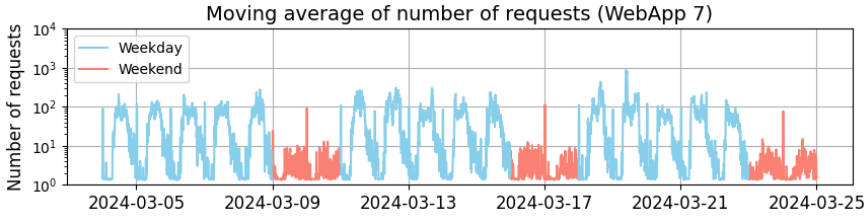

Figure 3.7: Web App: Line plot of the number of requests for Web app 7 over a period of three weeks

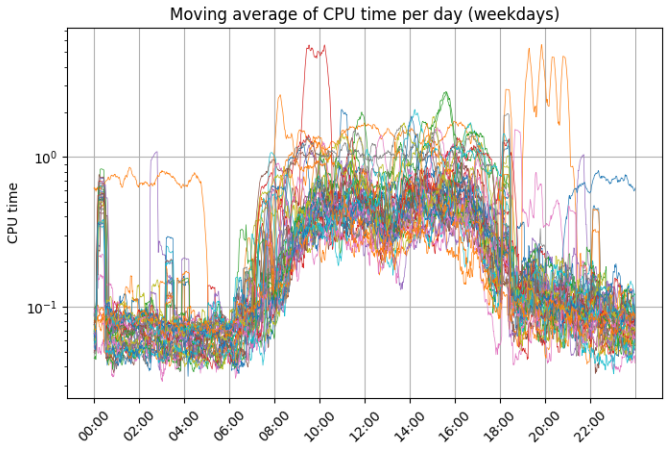

Figure 3.8: Web App: Moving average of CPU time per day for Web app 7

day. To further delve into these daily patterns, we visualise the CPU time per day on all weekdays for Web app 7 in Figure 3.8. The width of the moving average window is 20 minutes, as this clearly showed the relevant pattern while not smoothing over the volatile behaviour as much.

We can see that the pattern that was visible earlier in Figure 3.6 is a pattern that is quite apparent in all weekdays. Therefore, we expect that adding the input features regarding the time of day and day of the week will add to the predictive power of the model for Web app 7 in any case.

3.4 Production data

Later in this research, we estimate the performance of the models on data that the models were not trained on. We do this by evaluating the forecasting models on four Web apps from a production environment. We obtained data from four Web apps that are hosted in a completely separate Azure environment. For all four Production Web apps, a full month of data was available. For consistency, the full dataset was scaled using the same values as the original datasets, namely the means and standard deviations from the training set. This means that the mean and standard deviation of the data here are not necessarily approximately 0 and 1 respectively.

In order to possibly explain future results, we visually explore the production data as well. Figure 3.9 shows the correlations between the target and input features for both the production and non-production datasets. Correlations that did not have a value over 0.3 for either the production or non-production dataset are hidden.
As we can see, there are some strong correlations present in the production data that were not present in the data for the other Web apps. For the production data, the CPU time is highly correlated with both the IO Read and IO Write speeds, while there exists no correlation between these features for the Non-production Web apps. In contrast, there exists a correlation between the CPU time and the number of requests for only the non-production Web apps. The relatively weak correlation between CPU time and the amount of ingested data exists for both groups of Web apps.

Figure 3.10 gives the histogram of the CPU time and memory usage for the Pro-

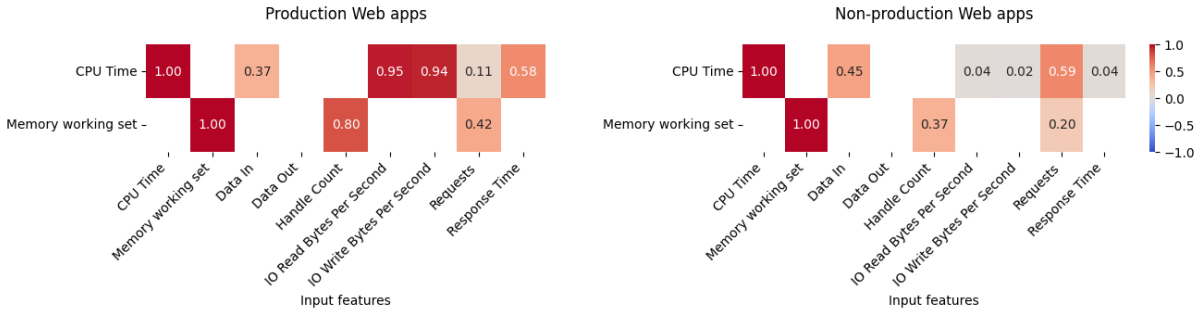

Figure 3.9: Production Web apps: Correlations between the target and input features

Figure 3.10: Production Web apps: Histograms of target variables for production Web apps

duction Web apps.
The plots shows similar patterns for the production Web Apps, compared to the non-production plots as seen in Figure 3.4. However, some differences in scale and shape are present between the two groups. For instance, production Web App 4 shows higher memory usage than all other Web Apps in the non-production group. Additionally, production Web App 2 shows significantly higher CPU time than most others.

3.5 Input and output data

In this section, we discuss different aspects of the input and output data for the forecasting models.

3.5.1 Input and output features

If we recall, the features that will be predicted are the features that are directly related to the computational units of the cloud resource. For the Web app, the predicted features were the CPU time and RAM usage. The input features are all the logged variables, together with the extra features discussed in Section 3.2.3. All of the present feature values, including the predicted variables, can be used as input features. As we are forecasting future values based on historical values, the predicted variables can be

used for forecasting future values of the predicted variables, potentially possessing high predictive power.

3.5.2 Output length

As discussed in section 2.4.4, the length of the prediction window can be seen as how important it is that the predicted computational load is met. This strongly depends on the nature of the tasks which the Web Apps of the App Service Plan are performing. As this information is not readily available for the Web Apps in this research, we arbitrarily chose the output length to be equal to three minutes.

3.5.3 Input length

Choosing the length of the input for the forecasting model is a trade-off between the amount of information that is available to the model and computational complexity, due to the increased number of computations that the model needs to perform. We expect that the few latest observations will be the most valuable for the model to base the predictions on. As a baseline, the input and output length were chosen to be of the same size. Experimenting with the training duration of multiple input sizes, resulted in choosing an input size of 12 observations or minutes. This way, there should be enough information available to the model, while still having acceptable total training times.

3.6 Conclusion

To summarize, there are many relevant computational metrics available about the usage of the different resource types. After inspecting the data, we cleaned and preprocessed the data accordingly. The cleaned data was explored using many different types of visualisations, revealing interesting patterns and relations with other variables. The variables that will be predicted by the forecasting models, are the variables that directly correspond to the usage of the computational units. As the predicted features for the Azure SQL databases suffered from an extreme feature imbalance likely due to inactivity in the cloud environment, the modelling chapter will focus on just the Web apps. The models that will be trained, will be evaluated on both a test set and a dataset from a production environment, which showed different patterns than the original datasets. The next chapter will discuss the modelling using the cleaned and preprocessed data.

Chapter 4

Modelling and Results

In this chapter, we discuss the choices that are made during the modelling part of this research and the first results that follow these choices. In Section 4.1 we describe the hyperparameter optimization of the Machine Learning models. In Section 4.2 we discuss relevant aspects of the training of the forecasting models and the prediction performances. In Section 4.3 we explore the different aspects, choices and results for the scaling model.

4.1 Hyperparameter tuning

As we recall from Chapter 2, both the RNN and LSTM have the same hyperparameters, which are the following:

- Number of layers
- Number of units per layer
- Dropout rate
- Learning rate
- Activation functions
- Optimizer

As the hyperparameter search space grows exponentially in the number of hyperparameters, choices need to be made regarding what hyperparameters will be searched over during the hyperparameter optimization. As the activation function and optimizers are less problem dependent than the other hyperparameters, they will not be included in the hyperparameter search, but rather be chosen beforehand based on related work. As the RNN is the baseline model, more effort will be put into selecting the activation function and optimizer for the LSTM, in comparison to the RNN.

For each run of the hyperparameter optimization, the relevant model must be initiated with the corresponding hyperparameters and trained on the training data, while being validated using the validation data. As a single training run does not take as much time, the total training time is mainly determined by the extensiveness of the hyperparameter search.

4.1.1 Recurrent Neural Network

With the philosophy in mind that over-tuning or over-optimizing a baseline model would defeat the purpose of a baseline model, a simple grid search over a wide range

of hyperparameters was performed. Using trial and error, the range of the hyperparameters were adjusted so that the optimal values were not consistently at the borders of the parameter ranges, indicating that the models were either underfitting or overfitting on the training data. The lower bound for the learning rate was however slightly restricted, as further decreasing the learning rate significantly increased training time.

The activation function that is used for the baseline model, is the ReLU function, as this is a widely used function capable of constructing complex, non-linear output functions. The Adam optimizer is used, as this optimizer is also widely used and well performing. More effort will be put into selecting the hyperparameters for the LSTM model, as the goal is to obtain the best performing model, and hyperparameter values play an important role in model performance.

4.1.2 Long Short-Term Memory

The activation function that will be used for all layers of the LSTM model is the hyperbolic tangent (tanh) function, as K. and K. [13] showed that LSTM models with this function generally outperform LSTM models with other widely used activation functions, such as Rectified Linear Unit (ReLU) and sigmoid.

The optimizer that will be used for the LSTM, is the Nesterov Adam optimizer (Nadam). Le et al. [16] investigated the performance of different optimizers for an LSTM model, and found that Nadam outperformed all other investigated optimizers in detecting network intrusion.

Using the hyperparameter optimization framework Optuna [2], an advanced search will be conducted over the remaining hyperparameters. This framework allows us to define suitable ranges per hyperparameter that Optuna will navigate through, searching for optimal combinations. This is done using complex algorithms that use historical run performances to find promising hyperparameter combinations. The extensiveness of the search is determined by the number of trials, or the number of models that are trained during the hyperparameter search. Choosing the number of trials is a trade-off between the degree of exploration of the hyperparameter space, and computational complexity. The hyperparameters that will be searched over using Optuna, are the four numerical hyperparameters, namely the learning rate, dropout rate, number of units per layer and the number of hidden layers of the model.

The last layer of both the RNN and the LSTM model consists of a flattened dense layer of size $n * m$, where n is the number of observations in the output time series, and m is the number of output features, which is $3 * 2$ in the case of the Web apps. This way, the output of the LSTM model can be mapped to the correct prediction format.

4.1.3 Hyperparameter ranges

For the RNN, the following hyperparameter ranges for the grid-search were determined using trial-and-error.

- **Number of hidden layers:** {1, 2}
- **Number of units per layer:** {16, 32, 64}
- **Dropout rate:** {0.1, 0.2, 0.3}
- **Learning rate:** {0.0003, 0.001, 0.003}

While the hyperparameter ranges for the LSTM were also determined using trial-and-error, Optuna was used to navigate over the hyperparameter space instead of

using a simple grid-search. As Optuna does not necessarily try all combinations of the hyperparameters like grid-search, bur rather intelligently nagivates through the search space, we can define broader ranges per hyperparameter. The following hyperparameter ranges were defined for the LSTM:

- **Number of hidden layers:** {1, 2, 3}
- **Number of units per layer:** {16, 32, 64}
- **Dropout rate:** Continuous value between 0 and 0.3
- **Learning rate:** Continuous value between 10^{-5} and 10^{-2} in logarithmic scale

4.2 Model training

Before the model is trained, the model weights are initialized using He normal initialization, which is a popular initialization technique used for initializing model weights in neural networks [7]. Furthermore, TensorFlow's implementation of this initialization technique can be made reproducible, which is an important part of creating reproducible models and results.

4.2.1 Early Stopping

During the training of both models, an early-stopping mechanism is used to halt the training process when there are no more significant improvements being observed. The training patience is set to 20, meaning that the training will halt when the validation loss has not increased in 20 epochs. As there is no limit set in the number of training epochs, the model will continue training until the training patience will run out. After the training epochs of a single model, the weights of the model that performed best on the validation set are restored, so that the best performing version of that model is obtained. This model is then stored on the machine that is training the models. This way, we can easily retrieve the model that performed best during the training phase.

4.2.2 Training time

The total training times it took for the RNN and the lSTM were approximately 21 hours and 10 minutes, and 42 hours and 1 minute respectively on a consumer-grade laptop. With a total of 54 evaluated RNN hyperparameter combinations, the average training run took approximately 25 minutes. For the LSTM, a total of 50 trials or training runs were performed, resulting in an average training run duration of approximately 50 minutes. The training of the LSTM was however limitedly parallelized, due to kernel errors when using a high parallelization grade in Optuna. During the training of both models, CPU parallelization was used, as this is implemented in the TensorFlow package. Implementing GPU parallelization can potentially decrease training time, but this was not tried during this research.

4.2.3 Final hyperparameters

Table 4.1 gives the hyperparameters of the RNN and LSTM model that performed best on the validation set during the training process. The model that had the best performance on the validation set will from here on be referred to as the best model.

Hyperparameter	RNN	LSTM
Number of layers	2	2
Units per layer	64	16
Dropout rate	0.1	0.108
Learning rate	0.0003	$1.66 * 10^{-5}$

Table 4.1: Hyperparameter values for the models that performed best on the validation set during the training process

As we can see, the number of layers and dropout rates for both models are equal and very close respectively. The units per layer for the RNN is 4 times higher than the LSTM. However, a single LSTM unit is more complex than a single RNN unit. Choosing a higher number of units per RNN layer could be seen as compensating for the decreased complexity per unit. The learning rate is also substantially lower for the LSTM than for the RNN.

4.2.4 Training performance

The forecasting performance metrics used in this research are the mean squared error (MSE) and mean absolute error (MAE). We use the MSE because it weighs larger errors more heavily than smaller ones, making it more sensitive to outliers and higher values. This metric is used as the loss-function during the training process. Next, we also evaluate the MAE due to the higher interpretability. Furthermore, comparing the value of both metric values can give us insight into the distribution of the errors.

Table 4.2 gives the training and validation loss values for the best RNN and LSTM models.

Loss value	RNN	LSTM
Training	0.3787	0.3553
Validation	0.1849	0.1572

Table 4.2: Training and validation loss values for both the RNN and LSTM models

As we can see, the LSTM outperforms the RNN on both the training and validation set by approximately 6% and 15% respectively. This is to be expected, as the LSTM is the more advanced model, for which more effort was put into obtaining the best hyperparameters. For both the RNN and LSTM model, the validiation loss is significantly lower than the training loss. A possible explanation for this is that the time series in the validation set are easier to forecast for the models. However, the training and validation loss values cannot be used to fully determine over- or underfitting, as they do not encapsulate the full training process.

To further investigate potential over- and underfitting, we plot the training and validation loss curves for both models. Figure 4.1 visualises the training and validation Loss and MAE per training epoch for the best RNN model.
The first figure shows a steadily decaying training loss, which is an indication that the model is learning the patterns that are present in the data. However, the validation loss curve lies significantly lower than the training loss curve, and is only decreasing during the first few epochs. The best validation performance is obtained at epoch 5,

Figure 4.1: RNN: Line plots of the training and validation MSE and MAE per training epoch

after which the validation performance increases slightly over the remaining 20 epochs. The validation MAE shows the same behaviour, decreasing for the first few epochs, after which it slowly increases. Even though the training loss is steadily decreasing, the training MAE only decreases marginally after epoch 5. A logical explanation for this is that the forecasting performance on relatively high values or outliers in the training set is increasing. This is because decreasing the error for extreme values has a stronger influence on the MSE than on the MAE, as the MSE scales quadratically with errors, while the MAE scales linearly.

Figure 4.2 shows the training and validation Loss and MAE per training epoch for the best LSTM model.
As we can see, all the curves for the LSTM are substantially smoother than the curves for the RNN. This might be due to the lower learning rate for the LSTM, as we saw in Section 4.2.3. A higher learning rate can cause the loss values to be more volatile, as weights are adjusted more significantly. Again, the validation loss curve lies significantly lower than the training loss curve, while the MAE curves are very similar. The best performing model was obtained at approximately epoch 240, meaning that the model was trained for a far longer period than the RNN.

Next, we investigate the validation performance per Web app. Figure 4.3 gives barplots showing the training and validation MSE and MAE of the RNN model per Web app. The y-scales have been adjusted so that they are equal in both plots.
Both the training and validation MSE show a significantly more variability between the Web apps than the MAE values. While for example Web app 1, 2, 5 and 6 have a lower MSE than their corresponding MAE, the contrary is true for Web app 4 and 7. This is an indication that there are different levels of skewedness in the errors per Web app. This can be caused by for example different patterns per Web app or the presence of outliers for some Web apps.

Figure 4.3 gives barplots showing the training and validation MSE and MAE of the LSTM model per Web app. The y-scales are equal in both plots, as well as equal to the y-axes in the previous plot.
There seem to be only small differences between the performance of the RNN and the

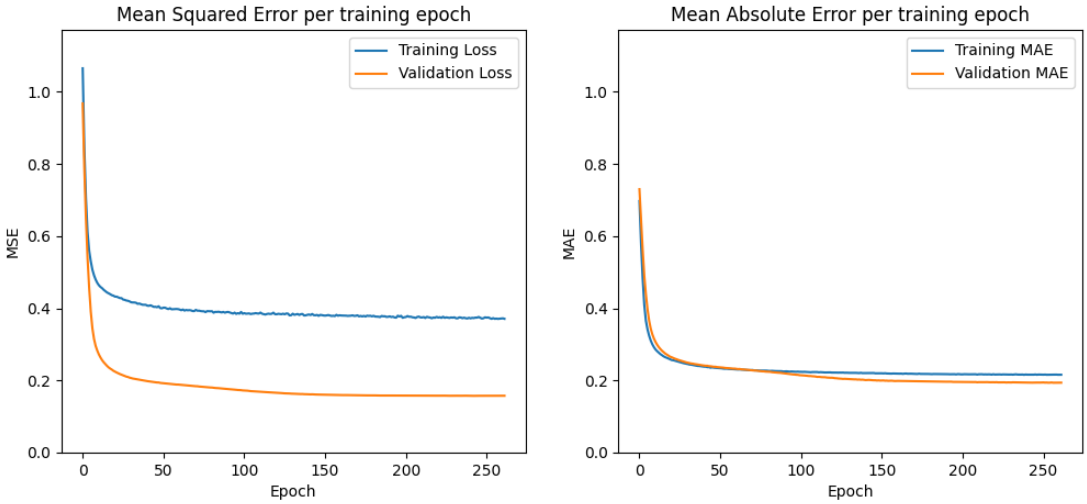

Figure 4.2: LSTM: Line plots of the training and validation MSE and MAE per training epoch

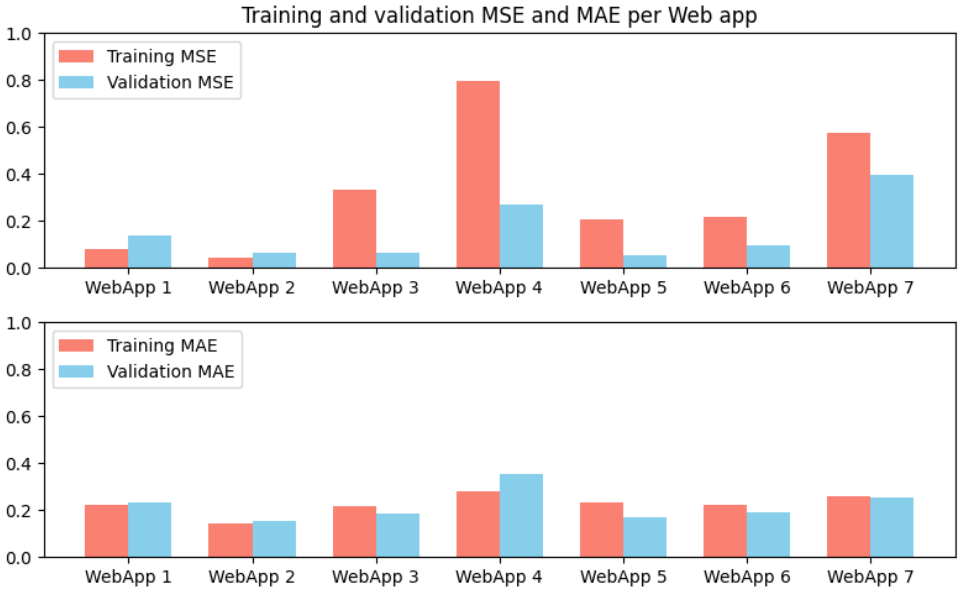

Figure 4.3: RNN: Bar plot of the training and validation MSE and MAE per Web app

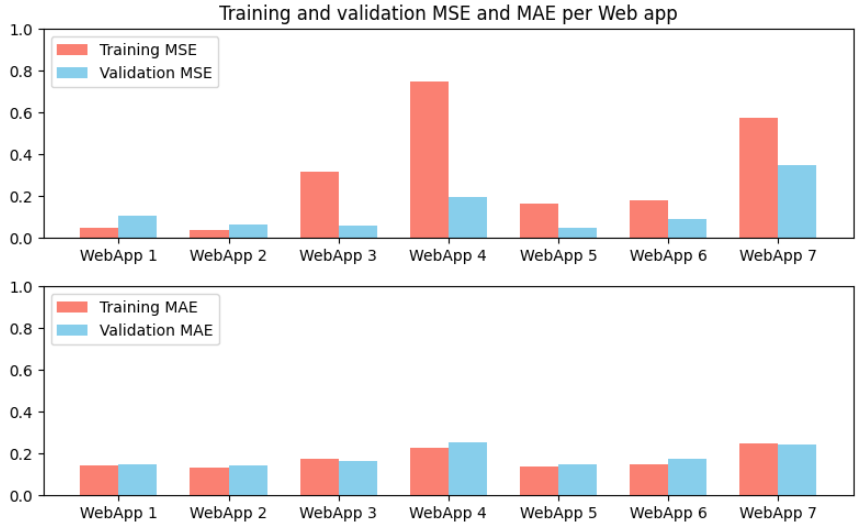

Figure 4.4: LSTM: Bar plot of the training and validation MSE and MAE per Web app

LSTM. For example, both the validation MSE and MAE for Web apps 1 and 4 seems to be slightly lower for the LSTM than for the RNN. Furthermore, the validation MAE is slightly lower for the LSTM than for the RNN.

Figure 4.5 shows the true and predicted CPU time and RAM usage for the validation set over an arbitrary period of approximately 40 hours for Web app 7. The predictions were made by the LSTM model.
As we can see, the predicted curves closely match the true curves for both the CPU time and RAM usage. The variability in the CPU time is however not fully accounted for in the predictions. There also seems to be a slight delay between the predicted and true curves. This is to be expected, as there is a window between the input observations and forecasting window, as discussed in Section 2.1.

To investigate the influence of using the Optuna hyperparameter optimization framework, we visualise the validation loss values over the number of trials in Figure 4.6. The grey lines indicate an improvement in the lowest loss so far.

As we can see, there are quite some improvements made over the different trials. While the later improvements are only marginal, it is still an indication that Optuna is successful in improving the validation loss. We can compare this with simpler methods, such as grid-search or random-search. As these methods do not account for any historical loss values or hyperparameter configurations, we can model this as randomly sampling from an arbitrary distribution. The probability of obtaining a smaller value than the minimum of n i.i.d. samples when drawing from a continuous distribution, is equal to $\frac{1}{n+1}$. Thus, for the simpler methods, we expect to obtain a better loss value with probability $\frac{1}{n+1}$ at trial n. When investigating Figure 4.6, we can see that we obtain a better loss value significantly more often using Optuna.

4.2.5 Production performance

To get an impression of the forecasting performance when the models would be deployed to a production environment, we evaluate the model on both the test set and

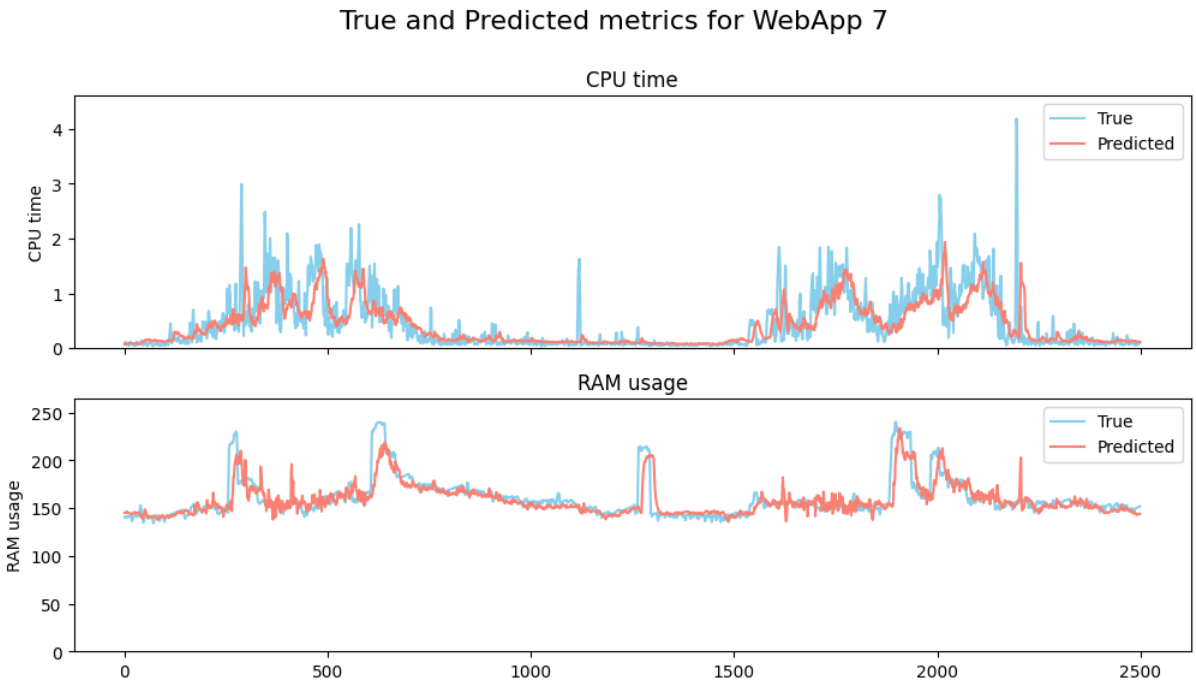

Figure 4.5: LSTM: True and predicted CPU time and RAM usage for Web app 7

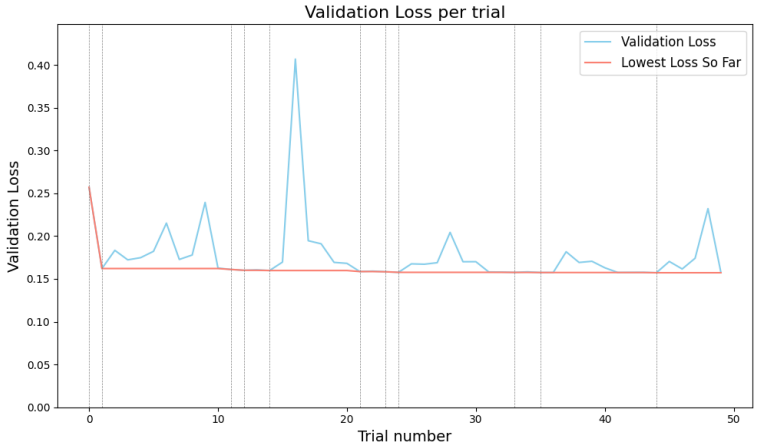

Figure 4.6: LSTM: Validation Loss per Optuna trial

datasets from an unseen production environment. The performance on the test set gives us an indication of how the model would perform in production for the Web apps on which the model was trained. Evaluating on the production dataset gives us an indication of how the model would perform in production when being used for Web apps that the model has not seen yet. Providing insight into the differences between these performances is very important, as these differences are an indication of how suitable a generalizing model, trained on data from a selection of Web apps, is for forecasting the computational load for other Web apps. A substantial difference between the performance on the test set and production set is an indication that the models perform significantly better on Web apps from which data was used to train the models. In this case, we could investigate training a model for each Web app, so that each model is trained on the data from the Web app for which it will eventually be used. The models that are trained for a single Web app, will be referred to as Specialist models.

First, we investigate the performance of the models on the test set. Table 4.3 gives the validation and test loss values for the RNN model.

Metric value	**RNN**
Validation loss	0.1849
Test loss	0.3579
Validation MAE	0.2272
Test MAE	0.2323

Table 4.3: Training and validation loss values for the RNN model

As we can see, the test loss is almost twice as high as the validation loss. A possible explanation for this would be that there can be different patterns present in the test set than there are in the validation set. However, as the test and validation MAE are very close, this is likely due to some high values or outliers in the test set that strongly influence the MSE, but modestly influence the MAE, as the MSE scales quadratic with the errors, while the MAE scales linear.

Next, we look at the performances of the LSTM model on the validation and test set, which are shown in Table 4.4.

Metric value	**LSTM**
Validation loss	0.1572
Test loss	0.3237
Validation MAE	0.1938
Test MAE	0.1991

Table 4.4: Training and validation loss values for the LSTM model

In this case, the test loss is more than twice as high as the validation loss, while the MAE values are still very close, indicating a potentially skewed error distribution. The test loss and MAE for the LSTM are approximately 10% and 15% lower than for the RNN, meaning that the LSTM will likely outperform the RNN when being deployed into production.

Next, we evaluate only the LSTM model on the production datasets, as it outperformed the RNN model on all performance metrics. In the following section, we

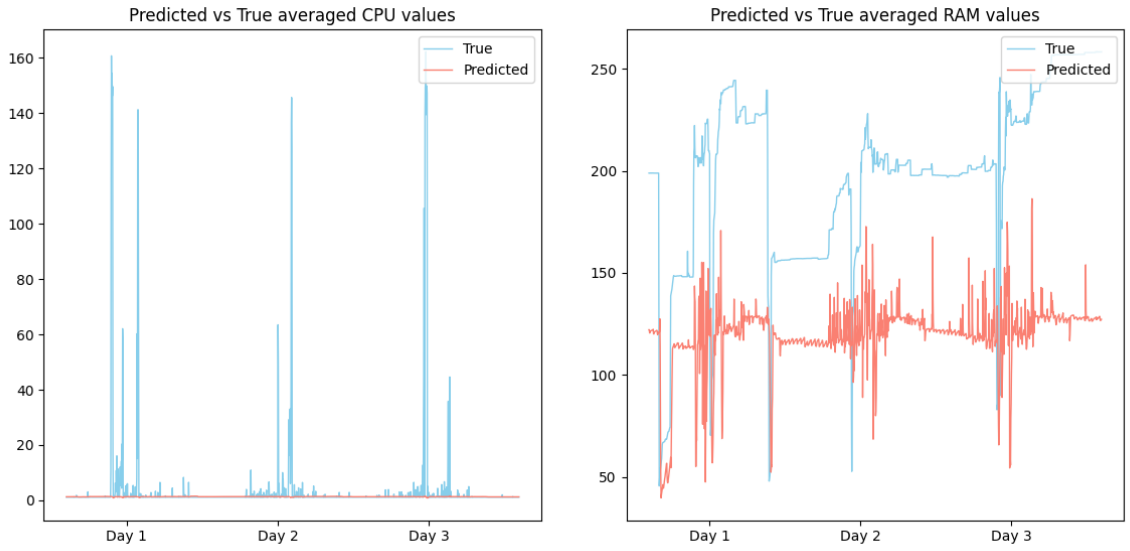

Figure 4.7: Predicted and true values of the CPU time and RAM usage of the production set

will build specialist models to investigate potential performance improvements. For training the specialist models, we split the production dataset into three sets: training, validation, and test. To compare the results of the generalist and specialist models, we use the same evaluation dataset, namely the test set of the production dataset.

Table 4.5 gives the production loss and MAE for the LSTM model on the production dataset.

Metric value	**LSTM**
Production loss	124.8940
Production MAE	1.7124

Table 4.5: Production loss values for the LSTM model

The production loss and MAE are approximately 400 and 9 times higher than the earlier obtained test loss and MAE respectively. While these difference can be considered extreme, they do not necessarily mean that the model is performing poorly on this dataset. As discussed in Section 3.4, the mean and scale of the production dataset can differ from the mean and scale of the original training dataset. This can cause the errors to have a different scale as well, while still performing relatively well.

Figure 4.7 shows the predicted and true values both the averaged CPU time and RAM usage for Production Web App 2, which showed the most clear patterns over time. Three consecutive weekdays are arbitrarily chosen as the visualisation window.

As we can see in the figure, the predicted values differ strongly from the true values for both the CPU time and RAM usage. For the CPU time, there is no clear pattern in the predicted CPU values. Furthermore, there seems to be a consistent offset or bias between the predicted and true RAM usage values.

The LSTM model, while outperforming the RNN on the datasets on which the models were trained, seems to perform poorly on the production datasets. This is an

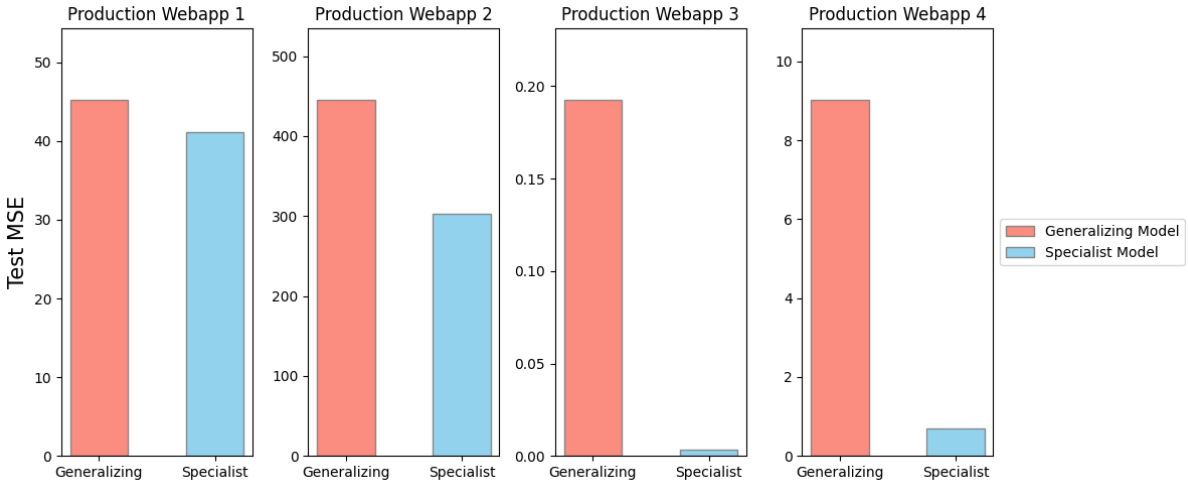

Figure 4.8: Test loss of the generalizing model and specialist models per Production Web app

indication that a model will perform better on resources from which data was used to train the model. This is logical, as the model will most likely be trained on similar patterns that will be forecasted in production.

One could argue that we could have retrained the generalizing model using training and validation set of the production data before we evaluated the model on the test set of the production data. This approach however raises complications when being deployed for an increasing number of Web apps. When the dynamic scaling mechanism is to be deployed for new Web Apps, the generalizing model must be retrained, also impacting the performance of the already present Web Apps. Furthermore, the model capacity must be suitable for the amount of data the model is trained on. When more new Web Apps are introduced, the initial model capacity can prove to be insufficient. We investigate another method of creating a scalable forecasting solution in the next section, namely specialist models.

So far in this research, we have only discussed the training process for two generalist models, meaning that a single model predicts the computational load for many different Web apps. As we discussed, these models performed poorly on data from Web apps on which they were not trained. Therefore, we expect that we can obtain performance increases for the Production Web apps when we train a single model, or Specialist model, on each individual Web app.

For the training process of the Specialist models, a configuration very similar to the configuration for the Generalist models is used. That is to say that the same hyperparameter ranges, batch size, improvement patience and number of training epochs were used. The only difference is that, to limit the computation time, we use only 20 Optuna trials for each specialist model.

Figure 4.8 visualises the test MSE per production Web app for both the generalist and specialist models.

The Specialist models outperform the Generalist model for all production Web apps. For the first two production Web apps, the decrease in loss values are approximately 9% and 32% respectively. For Web app 3 and 4, the loss values are approximately 56 and 13 times lower for the Specialist models than for the Generalist model. This is

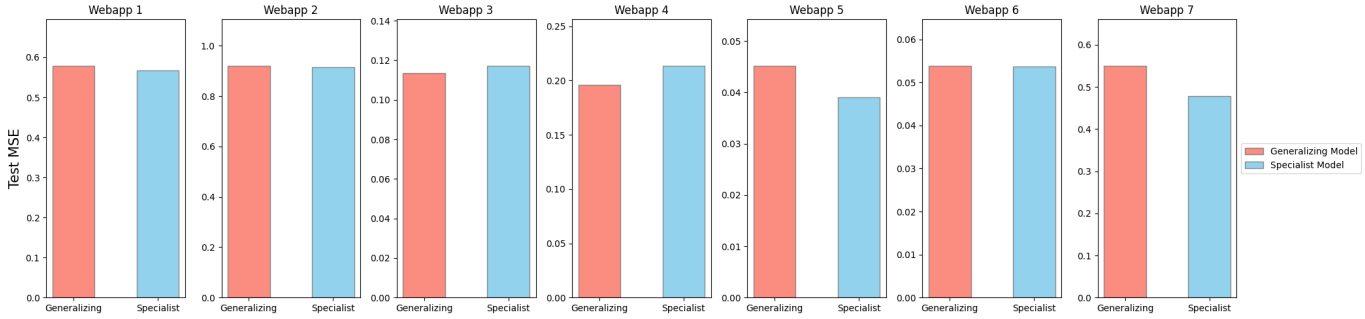

Figure 4.9: Test loss of the generalizing model and specialist models per Non-Production Web app

however to be expected, as the Specialist models forecast time series from Web apps on which they were trained, while the Generalist model does not.

Next, we visualise the performance differences between the Generalist and Specialist models for the non-production Web apps. This way, we can investigate whether performance increases are obtained when using a model per individual Web app, compared to a single model for the combination of these same Web apps. Figure 4.9 gives the test MSE per Web app for both the Generalist and Specialist models.

For Web app 5 and 7, the Specialist performance is slightly better than the Generalist performance. For Web app 7, this is to be expected, as the data from this Web app contained the most clear patterns out of all of the Web apps. When training a model on only these patterns, we expect the performance to be higher than when training the model on extra data that does not contain these patterns. For Web apps 3 and 4, the Specialist performance seems to be very slightly lower than the Generalist performance. A possible explanation is that both the Generalist and Specialist models managed to learn the patterns present in the data, and that the difference in performance is due to randomness during the training process.

An important thing to note is that we do not propose the Specialist models as the better option, compared to the Generalist model. We do however recommend using a model that was trained on all the Web apps for which it will be used to predict the load. While the forecasting performance of the Specialist models is generally similar or better, implementing and maintaining the Specialist models can be significantly more complicated and time-intensive, compared to a Generalist model. Choosing which type of model will be more suitable is dependant on many factors, such as the scale of implementation and the available work-hours to put into realising the implementation. This will ultimately be left to Info support. Advantages and disadvantages of both options are presented in Section 7.4.

4.2.6 Conclusion

To summarize, the Generalist LSTM outperformed the Generalist RNN on all performance metrics, with metric decreases ranging from 6% to 20%. The validation errors for both the RNN and LSTM were significantly lower than the MSE for the train and test set. This indicated that there were less high values or outliers present in the validation set. The LSTM seemed to correctly learn the patterns in the data of the non-production Web apps, being able to quite accurately forecast the computational

load. To investigate potential performance increases, we investigated the training of Specialist models. The Specialist models vastly outperformed the Generalist LSTM model for the production Web apps. For the non-production Web apps, the performance differences were very subtle, and most likely due to randomness. In the next section, we discuss the scaling model. This model takes as input the predicted CPU time and RAM usage, and decides whether the corresponding App Service should be scaled.

4.3 Scaling model

4.3.1 Suitable service tiers

To determine whether the App Service Plan underlying to the Web app should be scaled, we must first determine what computational tier we expect to be suitable for the resource. To achieve this, we must map the combination of the predicted metrics to a computational tier that satisfies both the CPU time and RAM usage. As discussed in Section 2.5.2, we can, for each metric, select the cheapest computational tier that satisfies the metric. When we union these tiers and take the highest tier of this set, we obtain the cheapest tier that satisfies all metrics. The scaling model will then use this tier as the target service tier. However, for the App Service Plans in this research, the lowest available service tier is almost always sufficient.

To be able to develop and test a realistic scaling model, the CPU time and memory usage will be multiplied by a fixed value. The App Service Plans that are considered in this research are all over-provisioned in terms of computational capacity, while having the lowest computational tier possible. Thus, investigating and implementing the scaling mechanism on actual computational tiers is not reasonable. For the current App Service Plans, they do not have to be upscaled and cannot be downscaled.

The lack of computational activity for all the Web Apps in this research might be representative of Web Apps in general. In conversation with field professionals, we found that the Web Apps that Info Support develops are mainly used for line-of-business applications, which are used within an organization. Consequently, these applications are generally not used as intensively as for example public websites accessed by users worldwide.

To make sure that we can still use the computational tiers used in practice while the patterns in the data are preserved, we multiply the CPU time and memory usage by a fixed value. This fixed value is chosen such that the scaled metrics span a large portion of the service tiers. After experimenting, we scaled the CPU time and RAM usage by multiplying them by 200 and 50 respectively. We first discuss the different service tiers, after which we will visualise the scaled metrics with the corresponding service tiers.

We assume that the available CPU time scales linearly with the number of CPU cores that are present in the relevant computational service tiers. That is to say, we assume that we for example have 60 and 240 CPU seconds available per minute when using a service tier with 1 and 4 CPU cores respectively. A hidden assumption when using the CPU seconds as a measure of computational capacity is that the CPU always runs on full capacity when CPU time is used. The different Premium service tiers and their corresponding number of cores, available RAM and cost are shown in Table 4.6. The number of assumed CPU seconds is added in the column for the number of CPU

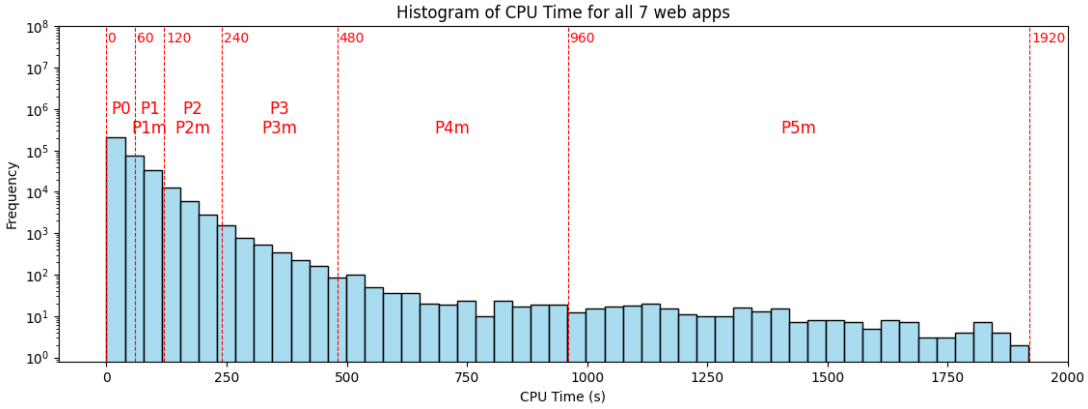

Figure 4.10: Histogram of scaled CPU Time with the corresponding service tiers

cores.

Service tier	Number of CPU cores	GB of RAM	Price per hour
P0	1 (60 s)	4	$0.210
P1	2 (120 s)	8	$0.330
P1m	2 (120 s)	16	$0.364
P2	4 (240 s)	16	$0.660
P2m	4 (240 s)	32	$0.728
P3	8 (480 s)	32	$1.320
P3m	8 (480 s)	64	$1.456
P4m	16 (960 s)	128	$2.912
P5m	32 (1920 s)	256	$5.824

Table 4.6: Amount of computational units and cost per service tier

As P5m is the highest service tier, all predictions higher than the available amount for P5m, will also be mapped to P5m.

Figure 4.10 visualises the scaled CPU time with the corresponding service tiers. For clarity, we omit the observations with a higher CPU time than the upper bound, which were 49 out of approximately 340.000 observations.
Due to the distribution of the CPU time, the majority of the observations fall into the first 3 tiers, namely P0, P1 and P1m.

Figure 4.11 visualises the scaled RAM usage with the corresponding service tiers. After experimenting with multiple scaling factors, we scaled the RAM usage such that the majority of the observations fall into the lower tiers. When scaling the features to also include the higher tiers, the RAM usage dominated the scaling decisions. This was caused by the distribution of the CPU time, which is significantly more skewed than the distribution for the RAM usage. For the RAM usage, substantially more observations would fall in the higher tiers relative to the CPU time.

4.3.2 Static service tiers

In Section 5.1.2 we will compare the test performance of the dynamic scaling mechanism to using a static service tier for the test set. To determine the most suitable static

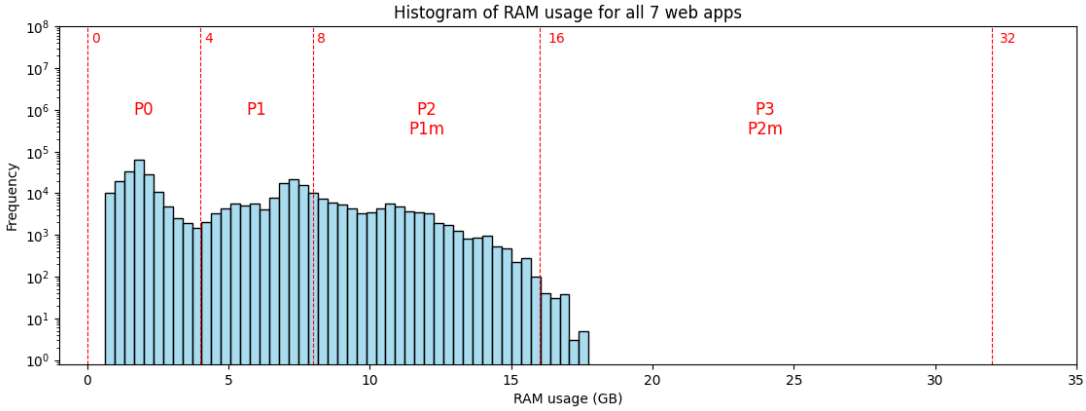

Figure 4.11: Histogram of scaled RAM usage with the corresponding service tiers

service tier for App Service Plan 1 and 2, we mapped the true CPU time and RAM usage to the corresponding service tier and inspected the corresponding histogram for the service tiers. These histograms are shown in Appendix A. In conversation with field professionals, we determined that the most suitable static tier for App Service Plan 1 and 2 were P1 and P4m respectively. The service tier that the Production App Service Plan is currently hosted in, is P2.

4.3.3 Results

In this section, we simulate and visualise the scaling mechanism over time, as described in Section 2.5.4. The model that will be used to make the predictions, is the Generalist LSTM model, as the predictive results of the Generalist model and Specialist models matched very closely. Figure 4.12 gives the averaged predicted and true CPU values from the test set, together with the upper bound of the CPU time over an arbitrary period of 10 hours for App Service Plan 2, in which Web App 7 is hosted.

In this figure, each grey line indicates a scaling decision. As we can see, the resource is being scaled quite often. While the amount of available CPU seconds is almost always above the predicted CPU time, it also matches the true CPU Time quite nicely. The scaling does seem to occur a little too late to meet the true demand when the CPU time periodically rises. This is logical, as there is also a gap, or scaling window, between the input data and the prediction window, as described in Section 2.4.4.

While the scaling decision is directly based on the predicted CPU values, not all of the predicted values are below the upper limit of the CPU time. During the scaling of a resource, it is possible that the forecasting model predicts a CPU time that is higher than the available resources for the tier that is being scaled to. The scaling model can however not cancel the current scaling operation in process and scale to the higher tier. Thus, in some cases, the predicted CPU values are higher than the available CPU resources.

Near the start of the period, we also see that the resource is scaling quite often, while the available CPU time remains the same. This is caused by the RAM usage around the 8GB level, as we will see in the next figure. The scaling occurs between the P1 and P1m service tier, which have different amounts of RAM, while having the same amount of available CPU time.

The true CPU time seems to be significantly more volatile than the predicted CPU

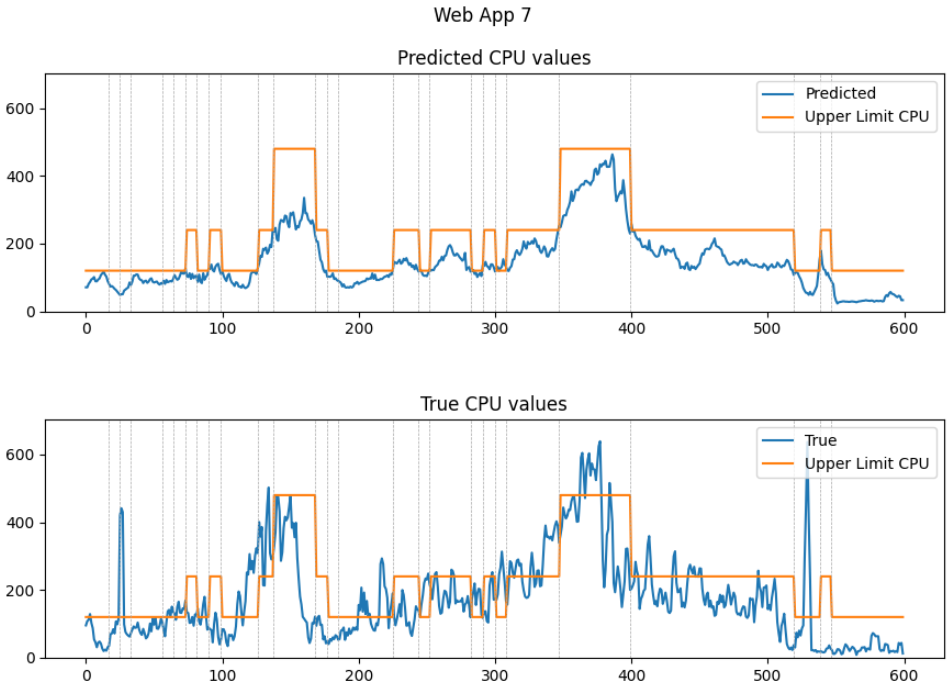

Figure 4.12: App Service Plan 2: Predicted and True CPU time, together with the available resources

time. While the predicted and true CPU time seems to match reasonably well, this volatility causes the true CPU time to exceed the available CPU time. In the next section, we will discuss a method to partly account for this volatility.

Figure 4.13 gives the predicted and true RAM usage, together with the available amount of RAM over the same period for App Service Plan 2.

Near the start of the period, we see that a lot of scaling occurs, as the predicted RAM seems to move around the border of 8GB. When surpassing he 8GB, the resource is scaled to the P1m service tier, which has 16GB of RAM. When going below 8GB, the resource is scaled back to the P1 service tier, with only 8GB of ram. As both service tiers have the same amount of CPU cores, the available CPU time remains constant, which we saw in the previous figure.

4.3.4 Capacity scaling multiplier

An issue that the current scaling model suffers from is that a service tier is selected when the forecasted load metric is even marginally lower than the upper limit of this service tier. For example when considering just the CPU time, when the forecasted CPU time is 950 s, the service tier P4m (960 s) is selected as the appropriate tier, even though the prediction indicates we use approximately 99% of the capacity. As we have seen in the previous section, the computational load forecasts are not perfectly accurate, potentially resulting in the true CPU time exceeding the threshold of the chosen tier.

We can make the scaling model more robust by multiplying the capacity per service

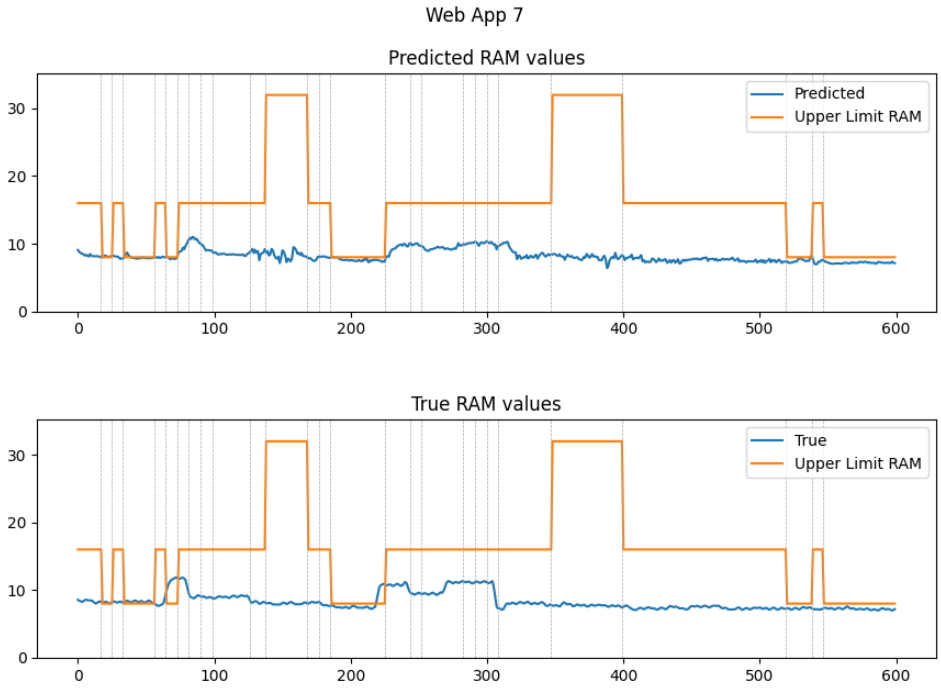

Figure 4.13: App Service Plan 2: Predicted and True CPU time, together with the available resources

tier by a fraction between 0 and 1. This capacity scaling factor indicates what fraction of the capacity we could at most use, based on the predicted load. Having a lower value for this factor causes the scaling model to scale to a higher tier earlier. This factoring constant can also be seen as a measure of how important it is that the future computational load is satisfied.

First, we investigate the influence and effectiveness of the capacity scaling multiplier by repeating the earlier steps using a scaling multiplier of 0.7. Table 4.7 shows the scaled capacities per service tier.
When using these adjusted service tier bounds, we expect that performance measures, such as the the fraction of time on which the resource is undersupplied, should improve. Figure 4.14 visualises the predicted and true CPU time with the available resources, using a scaling multiplier of 0.7.

While the resource is still being scaled a little too late, the amount of available resources seems to be able to handle the true CPU time better than without the scaling multiplier. Using this mechanism, we can account for a part of the combination of the volatile CPU time and prediction error.

4.3.5 Performance measures

As discussed in Section 2.5.3, we formulated multiple performance measures, to indicate how effective the scaling mechanism is. Table 4.8 gives the performance measures for both target features per App Service Plan. The scaling multiplier that was used is equal to 1.

Service tier	Number of CPU cores	GB of RAM
P0	1 (42 s)	2.8
P1	2 (84 s)	5.6
P1m	2 (84 s)	11.2
P2	4 (168 s)	11.2
P2m	4 (168 s)	22.4
P3	8 (336 s)	22.4
P3m	8 (336 s)	44.8
P4m	16 (672 s)	89.6
P5m	32 (1344 s)	179.2

Table 4.7: Amount of computational units per service tier, with a capacity scaling factor of 0.7

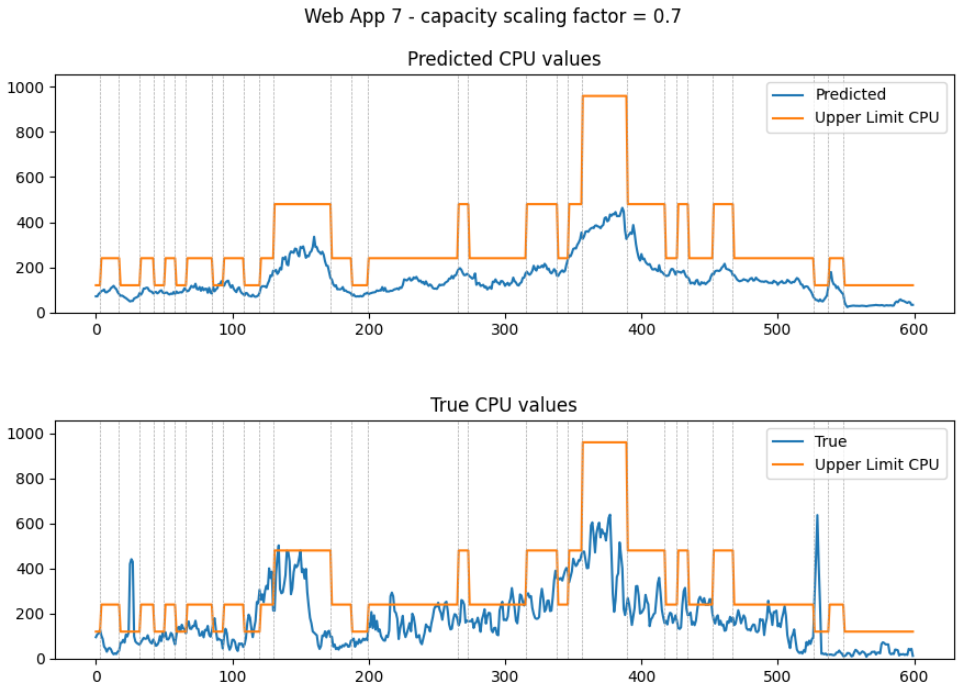

Figure 4.14: App Service Plan 2: Predicted and True CPU time, together with the available resources, with scaling multiplier 0.7

Metric	App Service Plan 1	App Service Plan 2
% of time underprovisioned CPU	15.9%	7.0%
% of time underprovisioned RAM	2.4%	5.6%
% lacking when underprovisioned CPU	13.0%	25.5%
% lacking when underprovisioned RAM	15.0%	6.2%
% utilization of resources CPU	71.3%	33.3%
% utilization of resources RAM	49.1%	79.7%
% of time spent scaling	13.6%	13.9%

Table 4.8: Performance metrics for both App Service Plans

For both App Service Plans, the percentage of time the RAM is underprovisioned, is lower than the percentage of time the CPU is underprovisioned. This is logical, as the RAM usage is significantly less volatile compared to the CPU time. Less volatility leads to smaller amounts of relatively high values or outliers, which inflate these metrics. The utilization of the CPU and RAM differs quite significantly per App Service Plan. This is most likely due to the different distributions of the target features, causing different scaling behaviour. The percentage of time the App Service Plans spent scaling are very similar.

Table 4.9 compares the performance measures of App Service Plan 2 for both multiplier 0.7 and 1.

Metric	No multiplier	Multiplier 0.7
% of time underprovisioned CPU	7.0%	3.4%
% of time underprovisioned RAM	5.6%	0%
% lacking when underprovisioned CPU	25.5%	27.3%
% lacking when underprovisioned RAM	6.2%	-
% utilization of resources CPU	33.3%	28.1%
% utilization of resources RAM	79.7%	47.8%
% of time spent scaling	13.9%	13.8%

Table 4.9: Amount of computational units per service tier, with a capacity scaling factor of 0.7

When using the scaling capacity multiplier, the percentage of time the resource was underprovisioned decreased for both features. This is to be expected, as the resource is upscaled earlier and downscaled later than when no multiplier is used. The percentage that was lacking when the resource did not have enough CPU time was however higher when using the multiplier. When using a multiplier of 1, many observations are slightly underprovisioned due to the volatility of the true CPU time. When using a multiplier of 0.7, we partly account for these observations, while the outliers remain. Thus, when using a lower multiplier, the outliers form a larger portion of observations in which the App Service Plan is underprovisioned, causing the percentage lacking when underprovisioned to inflate. As expected, the utilization of both the CPU and RAM is substantially lower for multiplier 0.7. The percentage of time the resource spent scaling are very similar.

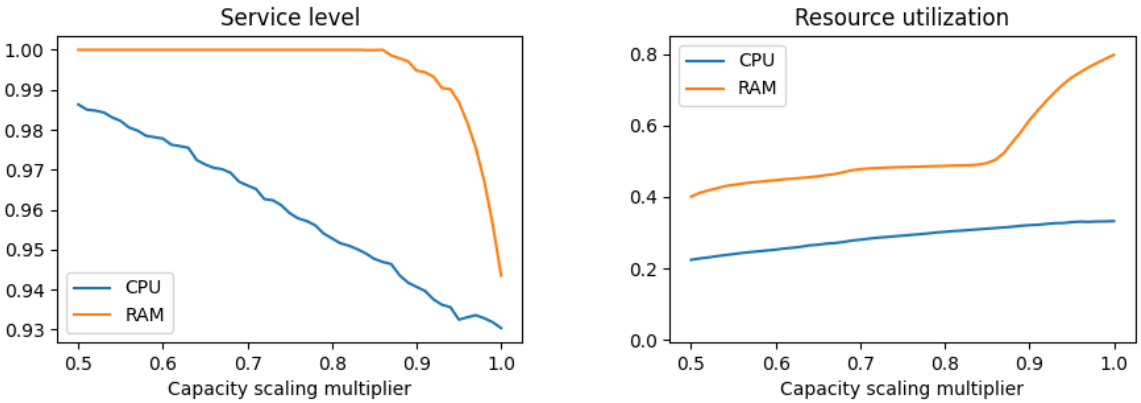

Figure 4.15: App Service Plan 2: Percentage of time underprovisioned and resource utilization against the capacity scaling multiplier

4.3.6 Multiplier tuning

Choosing the right capacity multiplier is a trade off between resource utilization and the percentage of time that the resource is underprovisioned. Choosing a lower capacity multiplier causes the resource to be scaled up earlier, resulting in a lower percentage of time that the resource is underprovisioned, while decreasing the average resource utilization. To numerically investigate this relation, we plot the percentage of time the resource was underprovisioned and the resource utilization against the capacity multiplier for App Service Plan 2 in Figure 4.15.
As expected, we see that there is a clear correlation between the capacity scaling multiplier and both the service level and the resource utilization. For the CPU time, the service level and resource utilization steadily decrease and increase respectively when increasing the capacity scaling multiplier. For the RAM usage, only relatively high scaling multipliers resulted in underprovisioning of the resource, which can be attributed to the low degree of volatility for this feature. Also, further increasing the scaling multiplier above 0.85 resulted in significantly better resource utilization. This is most likely caused by the distribution of the RAM usage. In this research, all relevant capacity scaling multipliers will be above 0.5. Next, we can define specific service level agreements and use the capacity scaling multiplier to achieve them.

4.3.7 Service Level Agreements

Service level agreements (SLAs) are generally established through conversations with the relevant customer and depend on various factors such as the tasks of the Web App, budget and customer requirements. Generally, Info Support uses one of three SLAs: 98%, 99.5%, or 99.8%. We can choose the suitable service level based on how critical it is for the RAM and CPU to be sufficiently provisioned.

We define the service level as follows:

$$\text{Service Level} = 1 - (\%\ \text{underprovisioned})$$

As insufficient RAM can cause the Web App to crash, ensuring there is sufficient RAM is crucial for the stability of the Web Apps. Therefore, we maintain an SLA of 99.8% for RAM. In contrast, having sufficient CPU power at all times is less critical for the Web Apps' tasks, as discussed with the responsible parties. Based on these discussions, an SLA of 98% was used for CPU power.

Table 4.10 presents the capacity scaling multipliers that achieve these SLAs for each App Service Plan.

App Service Plan	**Scaling Multiplier**
ASP 1	0.67
ASP 2	0.56
ASP Production	1.00

Table 4.10: Capacity scaling multipliers to achieve the defined SLAs per App Service Plan

Due to the lack of activity for the production App Service Plan, the service levels were achieved without using a capacity scaling multiplier, i.e. a multiplier equal to 1. For the RAM usage, a 100% service level was achieved for all App Service Plans when the shown scaling multipliers were used. In the next chapter, we compare the performance metrics when using the mentioned service levels, to Info Support's current approach.

4.3.8 Conclusion

At first, the scaling decisions seemed to be made slightly too late, resulting in numerous instances for which there was insufficient CPU time or RAM. The capacity scaling multiplier can be used to address this issue by limiting the expected percentage of used computational capacity, leading to earlier scaling to a higher service tier. Choosing a lower capacity scaling multiplier resulted in a significantly higher service level. Subsequently, we determined suitable service levels for both computational load features and their corresponding capacity scaling multipliers. Next, we will evaluate the results and compare the results to literature and current standards.

Chapter 5

Evaluation

In this chapter we compare and interpret the results from the modelling chapter. In Section 5.1 we compare the performance of the proposed methods to earlier research and the current approach in practice. Next, we explore the potential applications and interpretability of our methods in Sections 5.2 and 5.3 respectively. In Section 5.4, we assess the generalizability of the scaling solution. Finally, in Section 5.5 we examine the effectiveness of our approach in reducing total costs and energy usage.

5.1 Performance comparison

5.1.1 Literature

Similar studies that were found in literature had better numerical performance, which is most likely caused by a lesser degree of volatility in the target metrics. Zhang et al. [41] used an RNN to predict the computational workload in a Google cloud cluster, by using the CPU and RAM metrics over time. The lowest Mean Squared Error that was obtained for the standardised CPU and RAM were equal to $2.76 * 10^{-5}$ and $1.51 * 10^{-5}$ respectively.

In our study, the lowest MSE value was obtained for Production Web App 3, and was approximately equal to $3.4 * 10^{-3}$, which was significantly lower than the other MSE values in this research. Thus, the performance of this research was significantly lower than the mentioned study. However, in this research we predicted the computational load for individual resources, which is generally significantly more volatile than the computational load over a cluster of cloud resources.

Yadav et al. [40] predicted the hourly average load of a single distributed server in a cloud environment for an interval of 24 hours using LSTM models. The MAE that was obtained for the normalized load was equal to 0.043. For our research, the MAE was equal to 0.1991 for the normalized test set. However, we forecasted the minutely average load, which is a significantly more volatile metric compared to the hourly average load. Though, Yadav et al. [40] forecasted the computational load for a higher number of observations, making the comparison not fully representative.

5.1.2 Static service tier

In this section, we compare the dynamic scaling approach to using a single static service tier, which is Info Support's current approach for the App Service Plans in this research. For the dynamic scaling approach, we use the service levels discussed in Section 4.3.7.

The dynamic scaling mechanism yields significant performance utilization increases over the static service tier, while only slightly decreasing the Service Level.

First, we compare the performances of the static service tier and the dynamic scaling mechanism for App Service Plan 2. As discussed in Section 4.3.2, P4m would be the most suitable static service tier for this App Service Plan. Table 5.1 gives the relevant metrics for both scenarios.

Metric	**Dynamic scaling**	**P4m**
% of time underprovisioned CPU	1.9%	0.3%
% of time underprovisioned RAM	0%	0%
% lacking when underprovisioned CPU	30.4%	15.9%
% lacking when underprovisioned RAM	-	-
% utilization of resources CPU	24.4%	6.1%
% utilization of resources RAM	43.7%	6.2%
% of time spent scaling	11.2%	-

Table 5.1: App Service Plan 2: Amount of computational units per service tier, with a capacity scaling factor of 0.7

Dynamically scaling App Service Plan 2 resulted in a higher underprovisioning for the CPU, while significantly improving the resource utilization for both the CPU and RAM. The utilization of the CPU and RAM improved by a factor of approximately 300% and 605% respectively. The percentage of CPU time was lacking when the resource was underprovisioned was higher for the static service tier, which most likely again caused by outliers in the true CPU time. When dynamically scaling, the resource did spend quite some time scaling.

Next, we compare the dynamic scaling mechanism for the production Web apps to the current situation. The production Web apps are currently hosted in the P2 service tier. Table 5.2 gives the relevant performance metrics for both scenarios.

Metric	**Dynamic scaling**	**P2**
% of time underprovisioned CPU	0.8%	0%
% of time underprovisioned RAM	0%	0%
% lacking when underprovisioned CPU	33.1%	-
% lacking when underprovisioned RAM	-	-
% utilization of resources CPU	2.0%	.7%
% utilization of resources RAM	16.6%	4.2%
% of time spent scaling	0.6%	-

Table 5.2: Production Web Apps: Comparing performance metrics

As we can see, the underprovisioning for the CPU was minimal for the dynamic scaling mechanism, while there was no underprovisioning for the static service tier.

The utilization of the CPU and RAM resources increased by 185% and 295% respectively, compared to the static service tier. The utilization is still very low, as there is little computational activity present for the production Web Apps. As we can see in Appendix A, service tier P0 is the most prevalent service tier. As P0 is the lowest available tier, resource utilization cannot be significantly improved any further.

Next, we investigate the financial effectivity of the dynamic scaling solution. Here, we assume that billing is calculated per minute and only for the current service tier of the App Service Plan. This implies that scaling decisions do not incur any additional costs. The costs considered here are over the period of the test set, as shown in Table 5.3.

App Service Plan	Static tier costs	Dynamic scaling cost	Decrease in cost	Decrease %	Decrease per day
ASP 1	€32.06	€30,23	€1,83	5.7%	€0.45
ASP 2	€823.80	€159,26	€664,54	80.7%	€56,38
ASP Production	€94.84	€30.52	€64,32	67.8%	€10,74

Table 5.3:

For all App Service Plans, cost decreases were observed, ranging from 5.7% to 80.7%. For App Service Plans 2 and Production, the largest decreases were obtained, in both percentage and absolute cost. This is logical, as the potential cost savings during inactivity are larger for higher service tiers. In periods of computational inactivity, a relatively great decrease can be obtained by choosing the smallest service tier, compared to the high static service tier.

The percentage of saved energy is most likely strongly correlated with the percentage of cost saved. This is because the computational power costs are part of the costs that user pay when consuming computational power. Investigating the exact energy savings is outside of the scope of this research.

5.2 Applications

As the model generally scaled too late when the load suddenly increased, the proposed scaling solution may not be suitable for any type of application hosted using Web Apps. In the previous section, we saw that the model is incapable of predicting rises in the load beforehand, and rather responds to fluctuations in the load. This means that when the load suddenly rises, the computational demand is not met for at least a short period of time. Thus, the model is not suitable for applications for which it is vital that computational demand is always met, such as specific Healthcare applications. However, when it is vital that computational demand is always met, decreasing cost is generally a less relevant aspect.

Cloud resources that experience a quite consistent computational load are also not suitable targets for the dynamic scaling mechanism. First, the mechanism might not scale the resource enough to financially compensate the effort that was put into implementing the scaling mechanism. Furthermore, there is a risk that the load will oscillate around the border of two service tiers, causing the resource to be scaled excessively, potentially impacting availability.

5.3 Model interpretability

For both the RNN and the LSTM models, there is very little to no possibility to investigate the reasoning behind a certain prediction. The RNN and LSTM models

are special types of Neural Networks, which are based on complex numerical relations between the input features over time. Therefore, it is very challenging or impossible to gain insights into the reasoning of the model. The scaling model is however very transparent, as each mapping and decision can be fully explained.

One way to gain a limited insight in the behaviour of the forecasting models, is by investigating the outputs for given inputs. When slightly changing the inputs, the sensitivity of the model predictions can be assessed by investigating the response in the model outputs. Furthermore, the response of the model to for example sudden peaks or increases in computational load can be studied this way.

5.4 Generalizability

The performance of the scaling solution can most likely be generalized to other cloud resources, such as Virtual Machines and Azure SQL Servers. In this research, we predicted two computational load metrics, based on historical load metrics. As we have seen in Section 2.2.3, similar metrics are present for both DTU-based and vCore-based Azure SQL Servers. We expect that we can use the same methods, we can obtain similar performance for these resources, compared to the App Service Plans.

However, for some resources the excessive scaling can pose an issue in practice. When scaling a resource, a new compute unit is created and switched to once it is ready. For some resources, this process can terminate existing connections with for example users or background processes. Closing a connection can for example remove temporarily stored files and thus undo work, as well as requiring the user or process to reconnect to the resource. This would mean that the excessive scaling of a resource can have a significant impact on the usability of a resource.

5.5 Cost and energy usage

When deciding whether the solution can be used to successfully decrease cost and energy usage, the training, calling and maintaining of the models must be considered as well. As we saw in Section 4.2.2, obtaining the final Generalist or Specialist models can take a long time, and therefore use a lot of computational resources. From the moment the scaling solution is deployed, the solution will be called, or inferenced, each minute for each resource for which a scaling decision is to be made. Inferencing is significantly less computationally extensive compared to training the model, due to the reduced number of calculations that need to be performed. However, when the solution is deployed for a longer period of time for a large number of resources, the total inferencing cost may become significant in the cost and energy calculation.

The amount of consumption that is saved by dynamically scaling a resource is also dependent on the most prevalent service tiers for the resource. As we saw earlier in this chapter, the largest consumption decrease can be obtained by deploying the dynamic scaling mechanism for resources which are hosted using a high static service tier. For resources in a lower service tier, the training and inferencing of the models might limit the cost and energy consumption decrease, potentially causing it to only be effective in the long run, or even not effective at all.

Finally, choosing whether to implement a single Generalist model, or multiple Specialist models can also influence the cost and energy consumption. We will discuss this further in Section 7.4.

Chapter 6

Conclusion

In this research, we investigate dynamically scaling Azure Web Apps based on computational load forecasts using Machine Learning algorithms. Currently, the chosen service tier of Azure resources is based on the upper bound of the computational load. Furthermore, in practice, there is always a focus on ensuring there is enough capacity for a resource, rather than preventing overcapacity. Dynamically scaling Azure resources has the potential to significantly reduce both costs and energy consumption. For this research, the following research question was formulated:

> "To what extent can advanced analytics, such as Machine Learning, be used to predict cloud computation load in real-time and dynamically scale cloud resources to minimize cost and energy usage?"

Previous research in computational load forecasting has mainly focused on predicting the load for clusters of resources, or the averaged load for a single resource. For the scaling mechanism, previous research consisted of rule-based scaling mechanisms, which only allowed iterative up- or downscaling. This study differentiates itself by forecasting the computational load for individual resources, while using a tier-based scaling model. This scaling model maps the predicted computational load to a suitable target tier, after which it scales to that tier.

This research focused on Azure Web Apps hosted in App Service Plans (ASP), for which many computational load metrics were present, such as CPU time, RAM usage and latency. The data was logged per minute and during periods of 20 to 60 days, depending on the Web app. One of the ASP was hosted in a production environment, while the other two ASPs were not. All load metrics that were available were used as input for the forecasting model, while only the CPU time and RAM usage were forecasted for a period of three minutes. For one of the non-production ASPs, there was a clear difference between the patterns during weekdays and weekends. Furthermore, there was a clear pattern in the CPU time and number of requests through the day. For the other ASPs, we mainly observed subtle activity, with some random peaks.

We implemented two models: a Recurrent Neural Network (RNN) and a baseline model, and a Long Short-Term Memory (LSTM) model. More effort was put into obtaining the best LSTM model, compared to the RNN, and it consistently outperformed the RNN across all performance metrics. We also deleveloped Specialist LSTM models, meaning that each model was trained on data from a single Web App. We compared their performance to the Generalist LSTM model, which was trained on multiple non-production Web apps. The performance of the Specialist and Generalist models did

not strongly differ on Web apps on which both the models were trained. However, the Specialist models significantly outperformed the Generalist models for Web Apps on which the Generalist model was not trained. This indicates that predictions are more accurate when the models are trained on the specific data of the corresponding Web App. Despite this, we do not propose the Specialist models as the better models in this case, due to the higher implementation complexity compared to the Generalist models.

To evaluate the final effectiveness of our solution, we compared the dynamic scaling mechanism using service levels, to the current situation, which is using a static service tier. Our solution resulted in significantly higher resource utilization, while the percentage of time the resource was underprovisioned increased slightly. The cost savings that were obtained ranged from €0.45 to €56.38 per day per App Service Plan.

The solution did however suffer from excessive scaling. The amount of saved energy and cost will be dependent on the behaviour of the resource and the most prevalent service tiers. Furthermore, the proposed dynamic scaling solution will be suitable for resources that meet specific criteria. Examples of these criteria is that it is not vital that computational demand is met and that the demand is not extremely volatile.

To conclude, our dynamic scaling solution effectively predicted the computational load of cloud resources and scaled them accordingly, significantly outperforming current static approaches. However, the issue of excessive scaling remains a concern and could be detrimental when utilizing the solution in practice. Addressing this issue will require further research, as outlined in the next chapter.

Chapter 7

Discussion

In this chapter we reflect on the most important aspects and outcomes of our research. In Section 7.1 we discuss the limitations of this research. In Section 7.2 we outline potential areas for improvements and future work. In Section 7.3 we describe a possible architecture for deploying the dynamic scaling mechanism using Azure resources. In Section 7.4 we discuss the advantages and disadvantages of the Specialist and Generalist models. In Section 7.5 we provide recommendations for Info Support based on the concluded findings of this research.

7.1 Limitations

7.1.1 Exceeding available capacity

In the results, we saw that the consumed computational resources exceeded the available capacity of an App Service Plan, which in reality is not possible. In this research, we compared the CPU time and RAM usage with the upper bound of the service tier. We saw in the results that both features could exceed the available capacity at that time, which is impossible in practice. Thus, the actual CPU time and RAM usage would differ from the CPU time and RAM usage that was used in this research. In practice, the underprovisioning of the resource would potentially change the behaviour of the Web app, and introduce for example latency that cannot be easily adjusted for.

Thus, it is important that the training datasets of a Web app are gathered when the resource was overprovisioned. When the resource is underprovisioned during periods from which the training, validation or test sets are gathered, the true necessary CPU time for example will not be available, as it is upper bounded by the available capacity, rather than the necessary computational power. However, in practice, all resources are generally overprovisioned.

7.1.2 App Service Plan overhead

Due to operating system overhead, the amount of available resources the App Service Plan has available for the Web Apps is most likely lower than the assumed capacity [39]. The Virtual Machine underlying to the App Service Plan must run either Windows or Linux, which uses computational resources. This results in less CPU time and RAM available for the Web Apps. However, the amount of computational resources the operating system uses does not scale linearly with the computational capacity.

Thus, the overhead percentage is most likely smaller for App Service Plans with larger capacities.

7.1.3 Service levels

The presented service levels for the RAM and CPU of the App Service Plans are dependent on the time of day and the computational behaviour of the Web Apps. As we have seen in the results, service level violations generally occur due to sudden rises or peaks in computational load. When the computational behaviour of the Web Apps is more volatile and contains more rises or peaks, more service violations will occur as a result. Furthermore, the service level violations generally occur during periods for which the Web App is used relatively intensively. Thus, the service levels are likely higher during the moments for which it is important the service levels are not violated, namely the moments with higher activity.

7.1.4 Excessive scaling

The dynamic scaling solution led to excessive scaling of the target resource, which might be problematic in practice. Scaling decisions tend to occur during periods of high activity, further increasing their impact. For Web Apps, this can be particularly problematic, as processes or user sessions can be hosted in the memory available to the Web App. Scaling the resource causes the App Service Plan to switch to new memory, resulting in the sessions being dropped and users being disconnected. Approximately 14% of the time the resource spent scaling. If the resource undergoes frequent scaling throughout workloads, it is likely to significantly impact the usability of the resource. Other Azure resources, such as Azure SQL Servers and Virtual Machines suffer from this same issue.

7.1.5 Scaling costs

In this research, we assumed that scaling decisions did not incur any additional costs. However, if this assumption does not hold, the total cost could be significantly higher than presented. Investigations into the potential additional cost did not yield any valuable results. It is possible that there are extra costs associated with scaling decisions, such as paying for both the current resource instance, and the resource instance being created as a result of the scaling decision. The percentage of time that the resource is scaling would then be the percentage of time billed for both resource instances. Consequently, the total cost would then potentially be strongly influenced by the number of scaling decisions.

7.2 Future work

7.2.1 Interpretable forecasting models

For the models used in this study, there is no straightforward interpretation of the models decision-making processes. RNNs and LSTMs are known as black-box models, meaning that the forecasts result from complex numerical computations that cannot easily be interpreted. Using interpretable time series models, such as ARIMA models, can increase the trust in model predictions and allow for better debugging of results.

However, as these models are significantly less advanced, this would be a trade-off between interpretability and forecasting performance.

7.2.2 Intelligent scaling agent

The performance of the dynamic scaling mechanism can potentially be drastically increased by using a more advanced scaling model. The current scaling model is not really a decision-making model, but rather a simple computational load mapping. Using for example a Reinforcement Learning (RL) agent as a scaling model can greatly increase the flexibility and quality of the scaling decisions. Such an agent can choose to cancel a scaling in progress. This can be valuable when there are short peaks in the computational load. The prediction model can then inaccurately forecast the load, causing a suboptimal scaling decision. When it becomes evident that the scaling decision was suboptimal, the scaling agent can cancel the operation. Furthermore, the number of scaling operations that are initiated can also be reduced. We could do this by introducing a penalty for each scaling decision the RL agent makes. This way, the agent would be motivated to minimize unnecessary scaling actions, resulting in more robust scaling decisions, reduced cost and increased resource stability. Zhong et al. [42] implemented a Q-learning agent that determined whether to scale or not to scale a Virtual Machine, resulting in decreased SLA violations by up to 20% to 30%.

However, introducing an additional AI-based model into the scaling solution can pose difficulty for guaranteeing service levels. The performance metrics of the scaling solution are then also dependent on the behaviour of the scaling agent, which can be difficult to understand and explain [9]. Reinforcement learning agents can also respond unpredictably when the agent is to decide on unseen patterns. Thus, the service levels based on available datasets will be presented with greater uncertainty.

7.2.3 Downscaling multiplier

Another approach to limit the frequency of scaling decisions is by implementing an additional margin for downscaling. Currently, the same threshold are used for both up- and downscaling, which can result in excessive scaling when computational load forecasts oscillate around a specific threshold. For example, consider a scenario where a scaling multiplier of 0.8 is used, while the current service tier is P1. The resource is upscaled to P2 if the predicted usage exceeds 80% of P1's capacity, or 40% of P2's capacity. We can then introduce a condition that prevents the scaling model from downscaling back to P1 until the usage drops below for example 30% of P2's capacity, which would be 60% of P1's capacity. We can consider this as using an upscaling multiplier of 0.8 and a downscaling multiplier of 0.6. This way, we can partially tackle the problem of the oscillating computational load.

7.2.4 History duration

Understanding the amount of historical training data necessary for accurate computational load predictions helps determine when the dynamic scaling mechanism can be effectively applied for a specific resource. The amount of historical training data is also related to the degree of computational complexity, or size, of the model. Larger models will likely overfit earlier on smaller datasets, while they will be able to more effectively learn the patterns present in larger datasets. Furthermore, using smaller datasets and

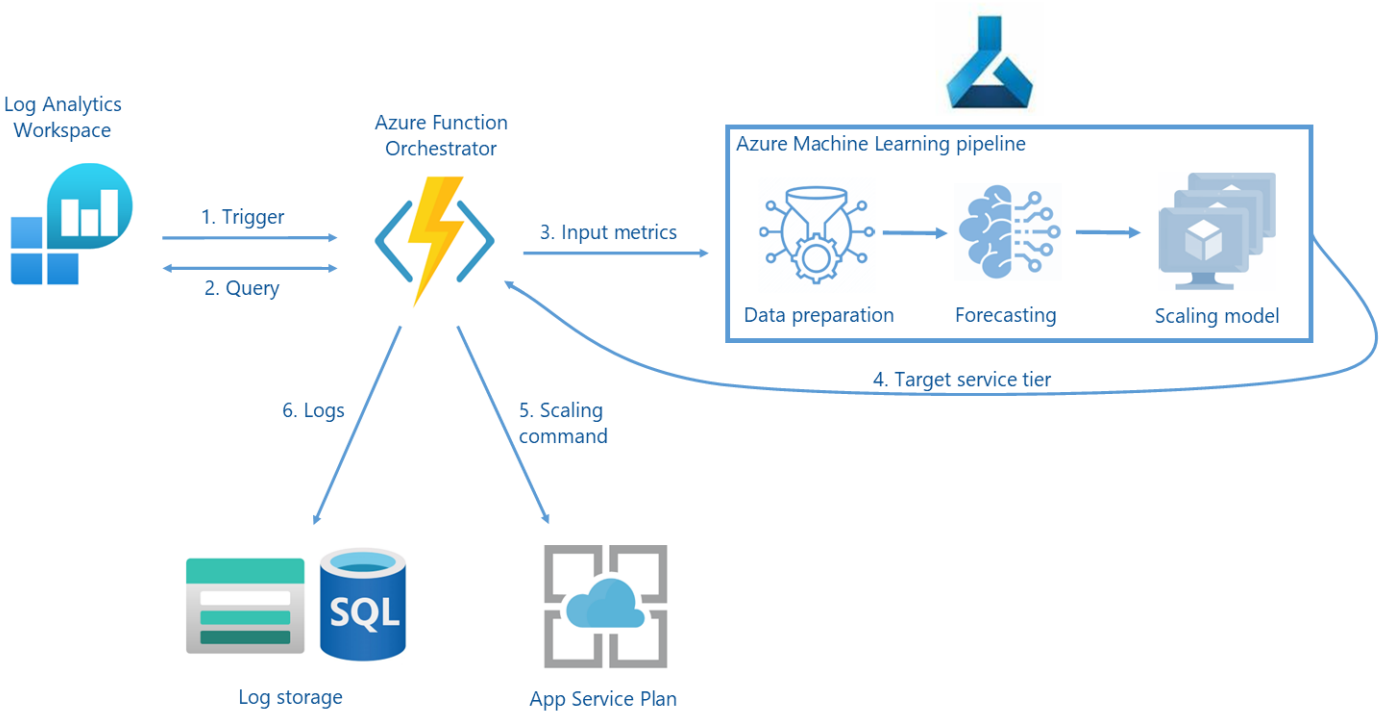

Figure 7.1: Architecture of scaling mechanism in Azure

thus models can have a positive impact on the performance of the models when they are retrained. When data shift occurs and the models are retrained, the models can more quickly adapt to new patterns in the data, due to the smaller model size.

For reference, Microsoft uses either one or two weeks of historical data for predictive autoscaling for Virtual Machine sets. In this research, the duration ranged from approximately 20 days to 60 days. Relating the model performance to the duration for which data was available was out of scope for this research. To investigate the impact of the duration on model performance, models can be compared by training them with different amounts of historical data.

7.3 Proof of concept

In discussion with field professionals, we designed a possible architecture for the dynamic scaling solution using Azure services. The Log Analytics Workspace (LAW) serves as both a metric-storage and a trigger for the dynamic scaling process. An Azure Function acts as the process orchestrator by handling the communication between the various services. All data and preparation and Machine Learning related tasks are managed within an Azure Machine Learning (AML) pipeline. Figure 7.1 gives an overview of the full mechanism and the used Azure services. The numbered arrows give the order and direction of communication.

The LAW collects and stores metrics per resource per minute. Each minute, a new record becomes available for each resource that is attached to the LAW. Whenever a new record becomes available for a resource, the orchestrator function is triggered.

When triggered, the orchestrator queries the LAW for the most recent observations for that resource. The orchestrator then registers relevant metrics, such as the name of the resource and the starting time. The query result is then sent to the Azure Machine Learning pipeline, in which the data preparation, forecasting model and scaling model are hosted.

When the input metrics are received by the Azure Machine Learning pipeline, they are first cleaned and preprocessed, as described in Section 3.2. The preprocessed observations are then used as input for the prediction model. The averaged prediction is then used as input for the scaling model. After the scaling model gives a service tier as output, relevant metrics such as forecasting duration and target service tier are communicated back to the orchestrator.

The orchestrator then sends the scaling command to the relevant resource, which will scale the resource. When the scaling command has been processed, the orchestrator can gather all relevant logs and save them to a database or storage account. The final scaling duration will then however not be present in the saved logs. This duration can be obtained later by using other Azure services, such as Azure Monitor Logs.

The dynamic scaling solution would most likely be hosted in the customer's Azure Subscription. This way, they are directly in control of the Azure resources without any dependency on other subscriptions. This means that the full mechanism should be deployed and maintained in the subscription of each customer for which the solution is to be used.

7.4 Specialist or Generalist

For inferencing, there is no significant challenge introduced when using Specialist models compared to Generalist models. For Generalist models, a single model must be inferenced to obtain the prediction using the input metrics. With Specialist models, the appropriate model must first be selected based on the resource for which a scaling decision is required. As this information can be made readily available, this poses no issue.

The resources required for inferencing and training a Generalist model are likely to be greater than those for the Specialist models. The Generalist model must be able to capture a wider variety of patterns, compared to the individual Specialist models. Therefore, the Generalist must be larger than the Specialist models in terms of computational complexity. However, training or inferencing a larger model requires for computational power per observation, as more calculations need to be performed, and more weights must be adjusted. Furthermore, Specialist models can be trained in parallel. As the Specialist are trained on separate datasets, training the models in parallel is trivial. Though, monitoring the training process for Specialist models can be more challenging than monitoring the training process for a single Generalist model.

However, versioning and maintenance could become problematic with Specialist models. For a Generalist model, only one model needs to be versioned. For Specialist models, each model would require individual versioning and updates. When the solution is deployed in customers' cloud environments, there would be multiple cloud locations, hosting multiple models, each with different versions. This complexity can make maintenance and management challenging.

As there are advantages and disadvantages to both Generalist and Specialist models, the final choice of the most suitable approach is left to Info Support.

7.5 Recommendations

Our first recommendation is to periodically investigate not only underprovisioning but also potential overprovisioning of resources. Currently, Info Support checks whether the requirements for resource capacities are determined and satisfied during project intake and annual check-ups. Expanding these checks to include overprovisioning assessments is likely the simplest method to reduce unnecessary costs and energy usage.

Our second recommendation is to investigate whether existing autoscaling methods already available in Azure can be utilized. Vertical scaling can introduce issues that affect the usability of a resource instance, so it is crucial to assess whether automatic vertical scaling is suitable for this task. Multiple horizontal scaling mechanisms are already available in Azure, which might be more suitable than the solution proposed in this study.

Our third recommendation is to investigate using a Reinforcement Learning-based agent as the scaling model in the proposed mechanism. An RL-agent can be used to tackle the most critical shortcomings of the proposed dynamic scaling solution, which was the excessive scaling of the resource instances that occurred.

Appendix A

Service tier histograms

In the figures below we show the histograms of the mapped service tiers for App Service Plan 1 and 2, and the production App Service Plan.

In conversation with field professionals, we determined that the most suitable static tier for App Service Plan 1 and 2 were P1 and P4m respectively. The service tier that the Production App Service Plan is currently hosted in, is P2.

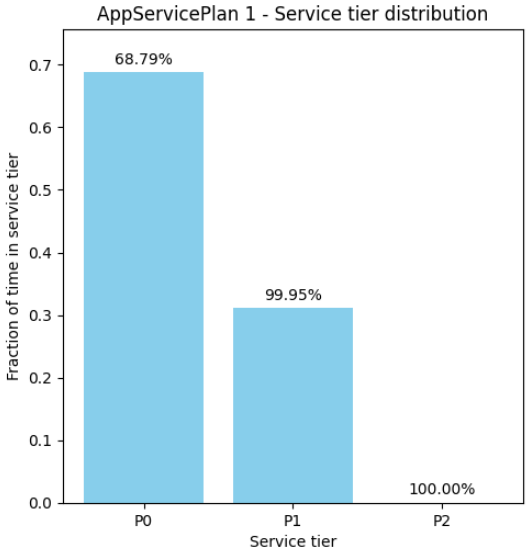

Figure A.1: Histogram of mapped service tiers for App Service Plan 1

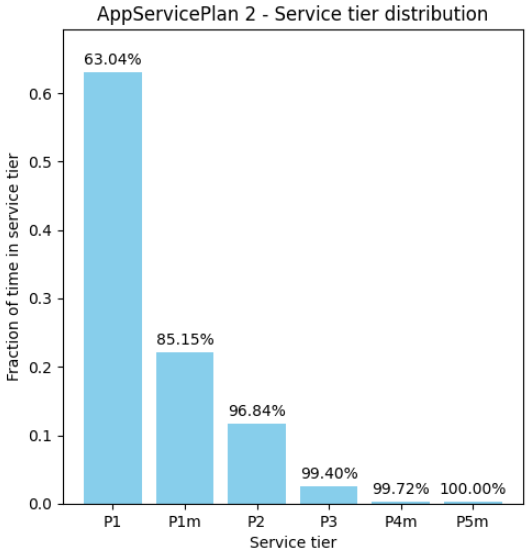

Figure A.2: Histogram of mapped service tiers for App Service Plan 2

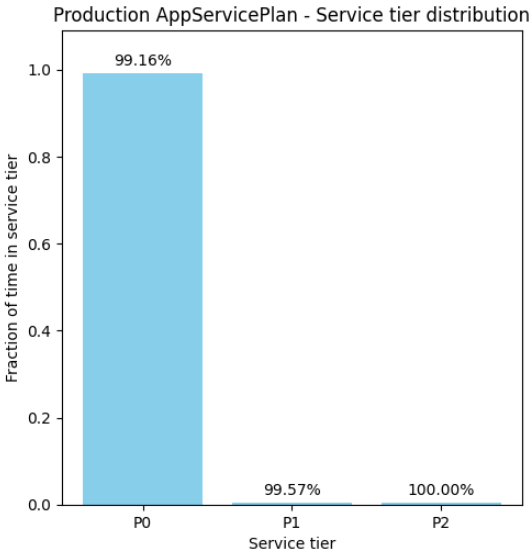

Figure A.3: Histogram of mapped service tiers for the Production App Service Plan

Bibliography

[1] Cloud Computing Market Size, Share | CAGR of 16.8%. URL https://market.us/report/cloud-computing-market/.

[2] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[3] C. Bergmeir and J. M. Benítez. On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213, May 2012. ISSN 00200255. doi: 10.1016/j.ins.2011.12.028. URL https://linkinghub.elsevier.com/retrieve/pii/S0020025511006773.

[4] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing*, 3(4):449–458, Oct. 2015. ISSN 2168-7161. doi: 10.1109/TCC.2014.2350475. URL http://ieeexplore.ieee.org/document/6881647/.

[5] M. Carroll, A. Van Der Merwe, and P. Kotze. Secure cloud computing: Benefits, risks and controls. In *2011 Information Security for South Africa*, pages 1–9, Johannesburg, South Africa, Aug. 2011. IEEE. ISBN 978-1-4577-1481-8. doi: 10.1109/ISSA.2011.6027519. URL http://ieeexplore.ieee.org/document/6027519/.

[6] G. Cheng, V. Peddinti, D. Povey, V. Manohar, S. Khudanpur, and Y. Yan. An Exploration of Dropout with LSTMs. In *Interspeech 2017*, pages 1586–1590. ISCA, Aug. 2017. doi: 10.21437/Interspeech.2017-129. URL https://www.isca-archive.org/interspeech_2017/cheng17_interspeech.html.

[7] L. Datta. A Survey on Activation Functions and their relation with Xavier and He Normal Initialization, Mar. 2020. URL http://arxiv.org/abs/2004.06632. arXiv:2004.06632 [cs].

[8] M. Duggan, K. Mason, J. Duggan, E. Howley, and E. Barrett. *Predicting Host CPU Utilization in Cloud Computing using Recurrent Neural Networks*. Nov. 2017. doi: 10.23919/ICITST.2017.8356348.

[9] C. Glanois, P. Weng, M. Zimmer, D. Li, T. Yang, J. Hao, and W. Liu. A Survey on Interpretable Reinforcement Learning, Feb. 2022. URL http://arxiv.org/abs/2112.13112. arXiv:2112.13112 [cs].

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] S. Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, Apr. 1998. ISSN 0218-4885, 1793-6411. doi: 10.1142/S0218488598000094. URL https://www.worldscientific.com/doi/abs/10.1142/S0218488598000094.

[12] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997. ISSN 0899-7667, 1530-888X. doi: 10.1162/neco.1997.9.8.1735. URL https://direct.mit.edu/neco/article/9/8/1735-1780/6109.

[13] V. K. and S. K. Towards activation function search for long short-term model network: A differential evolution based approach. *Journal of King Saud University - Computer and Information Sciences*, 34(6):2637–2650, June 2022. ISSN 13191578. doi: 10.1016/j.jksuci.2020.04.015. URL https://linkinghub.elsevier.com/retrieve/pii/S1319157820303505.

[14] S. B. Kotsiantis, D. Kanellopoulos, and P. E. Pintelas. Data Preprocessing for Supervised Leaning. 1(1), 2006.

[15] M. Law. Top 10 biggest cloud providers in the world in 2023, Feb. 2023. URL https://technologymagazine.com/top10/top-10-biggest-cloud-providers-in-the-world-in-2023.

[16] T.-T.-H. Le, J. Kim, and H. Kim. *An Effective Intrusion Detection Classifier Using Long Short-Term Memory with Gradient Descent Optimization*. Feb. 2017. doi: 10.1109/PlatCon.2017.7883684. Pages: 6.

[17] Z. Li, B. Gong, and T. Yang. Improved Dropout for Shallow and Deep Learning. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/hash/7bb060764a818184ebb1cc0d43d382aa-Abstract.html.

[18] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430, Honolulu, HI, USA, June 2012. IEEE. ISBN 978-1-4673-2892-0 978-0-7695-4755-8. doi: 10.1109/CLOUD.2012.103. URL http://ieeexplore.ieee.org/document/6253534/.

[19] Microsoft. What Is a Virtual Machine and How Does It Work | Microsoft Azure, . URL https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-virtual-machine.

[20] Microsoft. What is a SQL Database? | Microsoft Azure, . URL https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-sql-database.

[21] Microsoft. Scale resources - Azure SQL Database & Azure SQL Managed Instance, June 2023. URL https://learn.microsoft.com/en-us/azure/azure-sql/database/scale-resources?view=azuresql.

[22] Microsoft. Autoscale in Azure Monitor - Azure Monitor, Mar. 2023. URL https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview.

[23] Microsoft. Azure Virtual Machine Scale Sets overview - Azure Virtual Machine Scale Sets, Apr. 2023. URL https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview.

[24] Microsoft. VM sizes - Azure Virtual Machines, June 2023. URL https://learn.microsoft.com/en-us/azure/virtual-machines/sizes.

[25] Microsoft. Overview of autoscale with Azure Virtual Machine Scale Sets - Azure Virtual Machine Scale Sets, May 2023. URL https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview.

[26] Microsoft. Overview - Azure App Service, Aug. 2023. URL https://learn.microsoft.com/en-us/azure/app-service/overview.

[27] Microsoft. Scale up features and capacities - Azure App Service, May 2023. URL https://learn.microsoft.com/en-us/azure/app-service/manage-scale-up.

[28] Microsoft. Scale single database resources - Azure SQL Database, Jan. 2024. URL https://learn.microsoft.com/en-us/azure/azure-sql/database/single-database-scale?view=azuresql.

[29] M. K. Mohan Murthy, H. A. Sanjay, and J. Anand. Threshold Based Auto Scaling of Virtual Machines in Cloud Environment. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. Salinesi, M. C. Norrie, and O. Pastor, editors, *Advanced Information Systems Engineering*, volume 7908, pages 247–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-38708-1 978-3-642-38709-8. doi: 10.1007/978-3-662-44917-2_21. URL http://link.springer.com/10.1007/978-3-662-44917-2_21. Series Title: Lecture Notes in Computer Science.

[30] H. M. Nguyen, G. Kalra, and D. Kim. Host load prediction in cloud computing using Long Short-Term Memory Encoder–Decoder. *The Journal of Supercomputing*, 75(11):7592–7605, Nov. 2019. ISSN 0920-8542, 1573-0484. doi: 10.1007/s11227-019-02967-7. URL http://link.springer.com/10.1007/s11227-019-02967-7.

[31] T. M. P. Reader. The Staggering Ecological Impacts of Computation and the Cloud, Feb. 2022. URL https://thereader.mitpress.mit.edu/the-staggering-ecological-impacts-of-computation-and-the-cloud/.

[32] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507, Washington, DC, USA, July 2011. IEEE. ISBN 978-1-4577-0836-7. doi: 10.1109/CLOUD.2011.42. URL http://ieeexplore.ieee.org/document/6008748/.

[33] R. M. Schmidt. Recurrent Neural Networks (RNNs): A gentle Introduction and Overview, Nov. 2019. URL http://arxiv.org/abs/1912.05911. arXiv:1912.05911 [cs, stat].

[34] J. Shayan, A. Azarnik, S. Chuprat, S. Karamizadeh, and M. Alizadeh. Identifying Benefits and Risks Associated with Utilizing Cloud Computing. 3(3), 2013.

[35] A. Sherstinsky. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *Physica D: Nonlinear Phenomena*, 404: 132306, Mar. 2020. ISSN 01672789. doi: 10.1016/j.physd.2019.132306. URL http://arxiv.org/abs/1808.03314. arXiv:1808.03314 [cs, stat].

[36] R. H. Shumway and D. S. Stoffer. ARIMA Models. In *Time Series Analysis and Its Applications*, pages 75–163. Springer International Publishing, Cham, 2017. ISBN 978-3-319-52451-1 978-3-319-52452-8. doi: 10.1007/978-3-319-52452-8_3. URL http://link.springer.com/10.1007/978-3-319-52452-8_3. Series Title: Springer Texts in Statistics.

[37] T. A. B. Snijders. On Cross-Validation for Predictor Evaluation in Time Series. In M. Beckmann, W. Krelle, and T. K. Dijkstra, editors, *On Model Uncertainty and its Statistical Implications*, volume 307, pages 56–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988. ISBN 978-3-540-19367-8 978-3-642-61564-1. doi: 10.1007/978-3-642-61564-1_4. URL http://link.springer.com/10.1007/978-3-642-61564-1_4. Series Title: Lecture Notes in Economics and Mathematical Systems.

[38] B. Suleiman and S. Venugopal. Modeling Performance of Elasticity Rules for Cloud-Based Applications. In *2013 17th IEEE International Enterprise Distributed Object Computing Conference*, pages 201–206, Vancouver, BC, Canada, Sept. 2013. IEEE. ISBN 978-0-7695-5081-7. doi: 10.1109/EDOC.2013.31. URL http://ieeexplore.ieee.org/document/6658280/.

[39] G. Tong, H. Jin, X. Xie, W. Cao, and P. Yuan. Measuring and Analyzing CPU Overhead of Virtualization System. In *2011 IEEE Asia-Pacific Services Computing Conference*, pages 243–250, Jeju, Korea (South), Dec. 2011. IEEE. ISBN 978-1-4673-0206-7 978-0-7695-4624-7. doi: 10.1109/APSCC.2011.40. URL http://ieeexplore.ieee.org/document/6127969/.

[40] M. P. Yadav, N. Pal, and D. K. Yadav. Workload prediction over cloud server using time series data. In *2021 11th international conference on cloud computing, data science & engineering (confluence)*, pages 267–272. IEEE, 2021.

[41] W. Zhang, B. Li, D. Zhao, F. Gong, and Q. Lu. Workload Prediction for Cloud Cluster Using a Recurrent Neural Network. In *2016 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI)*, pages 104–109, Beijing, Oct. 2016. IEEE. ISBN 978-1-5090-5952-2. doi: 10.1109/IIKI.2016.39. URL http://ieeexplore.ieee.org/document/8281183/.

[42] J. Zhong, S. Duan, and Q. Li. Auto-Scaling Cloud Resources using LSTM and Reinforcement Learning to Guarantee Service-Level Agreements and Reduce Resource Costs. *Journal of Physics: Conference Series*, 1237(2):022033, June 2019. ISSN 1742-6588, 1742-6596. doi: 10.1088/1742-6596/1237/2/022033. URL https://iopscience.iop.org/article/10.1088/1742-6596/1237/2/022033.