Journal of Grid Computing (2024) 22:68
https://doi.org/10.1007/s10723-024-09783-1

RESEARCH

Enhancing Machine Learning-Based Autoscaling for Cloud Resource Orchestration

István Pintye · József Kovács · Róbert Lovas

Received: 18 June 2024 / Accepted: 9 October 2024

Abstract Performance and cost-effectiveness are sustained by efficient management of resources in cloud computing. Current autoscaling approaches, when trying to balance between the consumption of resources and QoS requirements, usually fall short and end up being inefficient and leading to service disruptions. The existing literature has primarily focuses on static metrics and/or proactive scaling approaches which do not align with dynamically changing tasks, jobs or service calls. The key concept of our approach is the use of statistical analysis to select the most relevant metrics for the specific application being scaled. We demonstrated that different applications require different metrics to accurately estimate the necessary resources, highlighting that what is critical for an application may not be for the other. The proper metrics selection for control mechanism which regulates the requried recources of application are described in this study. Introduced selection mechanism enables us to improve previously designed autoscaler by allowing them to react more quickly to sudden load changes, use fewer resources, and maintain more stable service QoS due to the more accurate machine learning models. We compared our method with previous approaches through a carefully designed series of experiments, and the results showed that this approach brings significant improvements, such as reducing QoS violations by up to 80% and reducing VM usage by 3% to 50%. Testing and measurements were conducted on the Hungarian Research Network (HUN-REN) Cloud, which supports the operation of over 300 scientific projects.

Keywords Cloud computing · Orchestration · Autoscaling · Machine learning · Proactive control · Metric selection · Resource optimization · Quality of Service (QoS)

1 Introduction

The wide adoption of virtualization has introduced the on-demand resource allocation and pay-per-use mechanisms to form and develop the cloud computing paradigm. The cost implication of resource utilisation has quickly led to the challenge of designing and developing efficient coordinated execution of multiple virtual machines to optimise costs for the owner. There

I. Pintye (✉) · J. Kovács · R. Lovas
Institute for Computer Science and Control, Hungarian Research Network (HUN-REN SZTAKI), Kende u. 13-17, Budapest H-1111, Hungary
e-mail: istvan.pintye@sztaki.hun-ren.hu

J. Kovács
e-mail: jozsef.kovacs@sztaki.hun-ren.hu

R. Lovas
e-mail: robert.lovas@sztaki.hun-ren.hu

I. Pintye · R. Lovas
Institute for Cyber-Physical Systems, John von Neumann Faculty of Informatics, Óbuda University, Bécsi út 96/B, Budapest H-1034, Hungary

J. Kovács
School of Computer Science and Engineering, University of Westminster, 115 New Cavendish, London W1W 6UW, United Kingdom

are a wide variety of orchestration solutions to centrally manage application with their particular auto-scaling and Quality of Service (QoS) requirements to utilise elasticity provided by cloud computing. These orchestration platforms are facing the challenge to meet the strict QoS requirements specified by the service owners while having little knowledge about the internals of the service behaviour and characteristics [1].

Since the internal characteristics of the applications and services are significantly affecting the resource utilisation, the scaling logic applied to control the underlying resources may differ from service to service. As a consequence it is a tedious task [2] for the service owner to specify and configure the scaling logic to reach the ideal scaling mechanism to optimise the resource utilisation as well as costs in parallel. To reach the optimal solution for a particular cloud service, one of the key elements is the knowledge of the operational characteristics of the service which is of course usually hard to discover due to its unpredictable nature of load and resource consumption.

Research results of different scientific fields can be utilised for regulation of resources. Each field approaches the given problem from a different aspect and places emphasis on different parts of the problem. The most relevant scientific fields are a) time series-based estimation and forecasts, b) control theory and c) machine learning. Each field and its results have its own advantages and there are specific cases and application areas where they operate more efficiently than the other ones. However, we found that each direction suffers from some boundary condition constraints, so our goal was to find a solution that eliminates application-specific constraints. Based on all these considerations the solution presented in this paper cannot be clearly classified under one scientific field (discipline), but we have adopted certain considerations from the fields of machine learning, time series analysis and regulation, combining these fields into an interdisciplinary solution.

For example, estimates based on expected load forecasting are difficult, if at all possible to provide correct scaling in situations where there is no recognizable pattern in the load generated by the incoming requests. Even, in situation where load is changing, while the number of incoming requests are close to constant, prediction is a real challenge for any scientific area. In case of time series based approaches the future trajectory of the incoming load is generally estimated and forecasted. Regulation based on the system's internal metrics establishes some relationship between the input metrics (state variables) and the response time. Machine learning plays a crucial role in trying to uncover these relationships based on (or learnt from) the observed data.

With the increasing utilisation of machine-learning algorithms for automatically learning and discovering the monitored behaviour, autoscaling methods have the potential to provide more sophisticated controlling over the cloud consumers (services) and resources in terms of provisioning. Machine-learning algorithms have the power of building knowledge about characteristics by monitoring the relation between the behaviour of the service and its continuously changing environment such as resource consumption and incoming loads.

In this paper we present our latest results about how we managed to improve an existing scaling algorithm towards a more sophisticated, optimised solution. We started with a detailed investigation of many scaling techniques and solutions in order to find candidate algorithm providing a good starting point in order to further develop it towards optimisation, proactivity and improved QoS protection. During our work we applied a collection of improvements to develop a more optimised controlling mechanism over the resources utilised by a cloud service.

As a result we elaborated a solution that is able to optimize the resource usage of cloud-based applications while keeping the QoS violations as low as possible. Our approach leverages data-driven machine learning techniques and controls for orchestration platforms. The proposed solution is thoroughly analyzed and investigated under various conditions. We uncover the potential weaknesses of the solution and outline possible directions for further research in this area.

After this brief introduction, the paper continues with a short overview of scaling techniques along with related works in Section 2. For a deeper understanding, in Section 3 we give a detailed description of the scaling algorithm considered as baseline to be improved in the rest of the paper. We introduce our proposed scaling approach in Section 4 giving a summary of the directions the baseline algorithm has been further improved towards. Section 5 dives into the details of the improvements related to metric selection techniques. The second collection of improvements is detailed in Section 6 where the focus is on proactivity and its effects on the entire orchestration mechanism. In Section 7 the behaviour of our proposed scaling algorithm has been investigated to find out how the algorithm performs in various situations. Finally, the paper concludes in Section 8.

2 Related Works

Zhong and Xu (2022) [3] give an excellent comprehensive study of the interrelation between autoscaling and machine learning techniques in cloud scaling. This survey effectively summarizes research findings and tendencies from 2017 to the present. However, our current work approaches the topic from a specific perspective that aligns more closely with our methodology, serves to understand our solution and place it within the broader context of scientific research.

Research in autoscaling of cloud-based systems is widespread [4], and has used many different approaches based on load forecasting, closed-loop control and the use of internal state variables. Biswas et al. (2015) [5] focused on predictive auto-scaling techniques in clouds with service level agreements, showcasing a proactive approach based on past workload predictions. Below, we present the most significant related works, categorized into three main groups: (1) methods based on load forecasting, (2) closed-loop control and related techniques, and (3) the use of internal state variables in scaling decisions.

A common characteristic of the studies we reviewed is the definition of a Service Level Indicator (SLI), such as throughput, response time, or resource utilization, and the attempt to maintain this SLO (Service Level Objective) while dynamically adjusting the number of resources. We also approach the issue from this perspective, and therefore, in our literature review, we only include works that take a similar approach.

2.1 Methods Based on Load Forecasting

Wang et al. (DeepScaling) [6] use spatio-temporal graph neural networks for cloud load forecasting. While this is sufficient to identify load patterns, their approach requires loads of high-quality data to match the model's accuracy.

Qiu et al. (FIRM) [7] suggest a fine-tuning resource management framework using Support Vector Machines and reinforcement learning. While this approach effectively handles static load patterns, it has limitations in dynamically changing environments as the authors acknowledge.

Previous work on load forecasting also includes Xu et al. (esDNN) [8], who apply deep neural networks for multivariate load forecasting. While both methodologies gain from load patterns predefined in the past, our solution goes further to work even in the face of unanticipated load fluctuations.

Imdoukh et al. (2019) [9] proposed a system that uses an LSTM model for accurate HTTP workload prediction. Toka et al. (2020) [10] use a similar approach but incorporated three different forecasting method: AR (Auto-regressive model), LSTM (Long Short-Term Memory), and HTM (Hierarchical Temporal Memory). They dynamically select the most suitable forecasting approach to match real-time request dynamics and accurately predict future load, measured in Queries Per Second (QPS). They consider not only the temporal data of incoming requests but also the effects of the time of day. Moreover, they contrast to the traffic in comparison to earlier periods, such as minutes before. They do not decide how many resources will act based on a machine learning model but calculate it using a rather straightforward mathematical operation: the predicted number of requests, QPS, opposes the number of requests that one container can handle according to the profile of an application.

Bansal and Kumar (2023) [11] also developed deep learning models to perform cloud load forecasting based on RNNs, CNNs, and autoencoder-based models. Their methodology addresses the key challenge of unpredictable end-user demand, which often leads to degraded service performance and revenue loss. The authors claim that their approach results in accurate predictions but only at large amounts of good quality data. In contrast, our approach also takes internal state variables into account, which can reduce the data requirements. The authors still underline that their approach can hardly estimate correctly the load patterns in case behavior of the system changes dynamically.

Yazdanian and Sharifian (2021) [12] introduce E2LG a hybrid LSTM/GAN model that decomposes temporal data series into different frequency bands and uses these to forecast expected load. The algorithm of E2LG, however, is very complex and computationally demanding, which may really limit it from working in real-time scenarios. Although the authors have thoroughly developed and tested the load forecasting algorithm (E2LG), they did not examine how an actual cloud-based system would scale based on these forecasts.

Patel and Bedi (2023) [13] use Multivariate Attention and Gated Recurrent Units (GRU). They improved the prediction accuracy on average compared to recent techniques that apply hybrid methods using LSTM and various other ANN models. The main drawback of their method is the large amount of data required to appropriately train these models.

2.2 Closed-loop Control and Related Techniques

Baresi et al. [14] propose a discrete-time feedback controller for containerized cloud applications. They start from the assumption that response time is a function depending on the number of resources and system load. Furthermore, they assume that when resources and load remain constant, the response time will converge to a steady value. Our method builds on the same assumption by integrating machine learning algorithms and we also assume that the dynamics of response time are not always linear.

Sabuhi [15] uses adaptive PID controllers backed by neural networks. In the paper performance analysis of these adaptive controllers evaluating a various loads and conditions, underlining applicability of control theory in resource scaling. Instead of using metrics that measure the system utilization, Sabuhi regulates resources (in his case, containers) based on error signal defined as a difference between real response time and desired response time (set-point). Input variables for a model estimating system performance are 1) the current number of resources being used and 2) the number of resources required. Output will be the response time of a system. Having made use of machine learning techniques (neural networks) the relationship between these variables is learnt. Our approach follows a similar principle for estimating system response time, but instead of focusing on the number of resources, we rely on system utilization metrics that describe the state of system.

Baarzi et al. [16] introduce the SHOWAR system, performing elasticity scaling with a control theory-based strategy. Their approach optimizes based on the process waiting time for CPU, but ignores telemetry data, such as memory usage and network traffic for more accurate scaling decisions.

Goli et al. [17] describe a machine learning model for automatic scaling of microservices, taking into account the dependencies between services. Their approach is close to our solution since their system evaluates and estimates the response time of a system what would be the impact of adding a new resource. This way, they determine how many resources their system requires. However, the authors consider only a limited set of system utilization metrics to train their model in an offline manner. As a result, it cannot adapt to the changing conditions in real time.

2.3 Utilizing Internal State Variables for Autoscaling Decisions

It can also make use of internal system state variables at its disposal like CPU usage, memory, and network traffic for autoscaling. All these help in describing or modeling the current state of a system such as an early example of this approach discussed by Abrantes and Netto (2015) [18]. This is a significant methodological shift with respect to previous approaches that try to predict future requests using time series models or neural networks based on incoming requests, and then use such predictions to adjust the number of resources accordingly.

Podolskiy et al. [19] utilize machine learning models to predict application performance indicators (SLIs) such as response time and throughput. The trained models aim to forecast these SLIs based on the resources allocated to applications (e.g., CPU capacity) and load levels (e.g., the number of concurrent requests). While their approach is well-suited for maintaining SLOs, they also assume a linear relationship between SLIs and resources. Podolskiy employs Lasso regression to create simpler models and reduce the risk of overfitting. In contrast, we use neural networks with dropout layers to mitigate the risk of overfitting.

Rossi et al. [20] provide a dynamic multi-metric scaling approach that learns how to adjust the thresholds using reinforcement learning. The basic idea is how threshold values of different resource usages, like CPU and memory, are set dynamically and automatically, rather than depending on a manually configured and hence always suboptimal fixed threshold. Their results, from simulations and prototype-based evaluations, demonstrate that the proposals have advantages over the current state-of-the-art. The RL models were learnt first in a simulator designed to approximate the behavior of cloud services. This simulator was very important during this initial phase of learning, which emulated the dynamic behavior of cloud services. The authors note that although their RL-based approach is effective, definite systems modeling is not an easy task, and treatment of several metrics complicates the convergence of the training. Also, in the case of their Deep-Q Learning-based solution, the right Q-network architecture happens to be an empirical process that requires a large number of iterations and may deserve pre-training for enhanced learning speed.

Zhang et al. [21] focus on resource management for cloud-based microservices through their framework called Sinan. They focus on QoS maintenance by taking into account or utilizing the service times of all microservices involved in composite services. It tries to make optimal decisions in resource allocations across microservices so that end-to-end latency targets are maintained and QoS violations are avoided. It will ensure an end-to-end latency of the next timestep across multiple time horizons using a CNN and provide resource utilization data on CPU, memory, and network usage (both in terms of packets received and sent). This kind of prediction helps proactive control over deterioration in performance even before it happens. Like in our proposed solution, their system is also based on a performance model aimed at predicting the expected response time; however, the real difference lies in how we determine and calculate the required amount of resources.

Nhat-Minh Dang-Quang et al. [22] discussed their method which forecasting the throughput of a network using a Bi-LSTM model. They do not feed the number of incoming requests - such as HTTP requests - into the model to make the prediction. Similar approach presented by Jayakumar et al. (2022) [23]. Prior to that, different telemetry data from the system itself are analyzed to make a prediction on network throughput using time-series analysis. They monitor network throughput but also several other related telemetry metrics in order to deduce the inference of future behavior of the network throughput. This is a fundamental change of methods trying to predict future request counts based on incoming requests with time series models or even neural networks. The authors note that although the system is performing very well with variables with network throughput and memory usage but more inputs metrics may be needed in the future to attain more accurate predictions.

2.4 Further Methods and References

In addition to the specific studies on predictive scaling and workload forecasting, recent survey research offers a broader perspective on the field. The survey by Pfeifer et al. (2023) [24] provides a critical analysis of both traditional statistical methods and modern machine learning techniques in the context of time series forecasting.

Yongkang Li et al. [25] present challenges and opportunities in serverless computing. The authors focus specially on resource usage threshold optimization and system scalability. Our solution is related by using more advanced predictive models and closed-loop control techniques that ensure the stability and scalability of the system even in serverless environments.

Hossen et al. [26] propose the PEMA system, which allows efficient microservice auto-scaling and QoS assurance of cloud-based microservices without seeking support from complex machine learning models. Their system adapts to time-varying workloads through an iterative feedback mechanism and is adaptive with regards to changing environmental conditions of microservices, including hardware or software updates. The authors present their PEMA results with detailed experimental results, but they also comment that the proposed PEMA system can not handle all complex dependencies between microservices. Moreover their solution does not result in absolutely optimal resource efficiency and there may be scenarios where the system will find suboptimal resource allocations during the search process and cases where this method is not entirely free from SLO violations.

2.5 Summary and Critical Analysis

The methods face several challenges, which have also been pointed out by the authors themselves.

It has been noted in several works that the complexity of the presented methods may be challenging, even in dynamically changing environments. For example Sabuhi [15] mentions the cumbersome parameterization of an adaptive PID controller. Baarzi et al. [16] mention the general complexity of the underlying theory of control. This is perhaps relevant more than ever in cloud environments, where the two prime requirements are fast adaptation and scalability.

Most of the methods require a great deal of representative data. For example, the fine-grained resource management framework presented by Qiu et al. [7] relies a great deal on large datasets and can be challenging in real-time applications. Wang et al. [6] point out that high accuracy of the predictive model depends highly on the quality and quantity of data.

Several authors underline that proposed methods do not always adapt very well in a rapidly changing environment. Rossi et al. [20] note that in case of reinforcement learning-based scaling, it is hard to update the thresholds dynamically. On his part, Podolskiy et al. [19] note that self-adaptive resource allocation may suffer from changing load patterns.

Some studies, such as [10], assume that the overall performance of scaled systems is directly proportional to the number of resources. However, this does not often reflect reality.

The above-mentioned authors also did not pay enough attention to the selection of appropriate and relevant state variables based on which the performance of the system can be well described and estimated.

3 Introduction of the Baseline Scaling Algorithm

The method discussed in Wajahat's papers "MLscale: A machine learning-based application-agnostic autoscaler" [27] and "Using machine learning for black-box autoscaling" [28] has been empirically validated, demonstrating its effectiveness in various environments. This lays a solid foundation for our decision to adopt this method as a benchmark for our research.

In the context of autoscaling for cloud environments, particularly for workloads that are highly unpredictable, choosing the proper approach is important to ensure both efficiency and performance. Many existing autoscaling methods address specific difficulties, such as fluctuating workloads or varying task complexities. However, few methods are comprehensive enough to address all of these issues together.

In order to fully understand the proposed orchestration approach, we consider the knowledge of this baseline algorithm [27, 28] important. The baseline algorithm aims to optimally and dynamically determine the number of cloud-based virtual resources to maintain the response time of an internet service within a predefined range. The algorithm consists of three main steps:

- applying a series of linear regression (LR) models to estimate the value of metrics caused by the change of resources
- applying a neural network to predict response time based on estimated metrics
- calculating the optimal scaling factor based on the predictions of possible scaling actions

In the next subsections, we give a detailed, formal explanation on the three main parts of the algorithm and an overall summary of the scaling procedure.

3.1 Linear Regression Models for Estimating Metrics

For each metric there is a separate model realised by simple linear regression. Each model learns how a given metric value changes in response to resource changes. The training of the model is performed during a period when scaling activities on the web application are periodically executed in a free-run to observe, measure and record the necessary metrics and their changes. It is important that at this point we do not consider how the changing of resource affects the response time, we only consider how it affects the observed metrics.

We use a linear regression model (g_{LR}) for each metric. These models estimate how a given metric changes in response to the change in the number of virtual machines (w) and the change in resources (k), where k, the scaling factor, represents the number of new resources and can be negative if resources are removed. It's important to note that the value of w, representing the number of virtual machines before scaling, can not be less than 1 and $(w + k)$ must be greater then 0.

For each metric m_i, we calculate the estimated post-scaling value the following way:

$$m_i' = c_{i,0} + c_{i,1} \cdot m_i \cdot w/(w+k) + c_{i,2} \cdot m_i \cdot k/(w+k) \quad (1)$$

where m_i' is the estimated value of the i-th metric after scaling, m_i is the value of the i-th metric before scaling, w is the number of virtual machines before scaling, k is the value by which we want to change the number of virtual machines. The $c_{i,0}, c_{i,1}, c_{i,2}$ are the linear regression coefficients for the i-th metric, determined from training data.

To determine the estimated metrics for each k scaling, we use the following equation:

$$m_i' = g_{LR_i}(m_i, w, k) \quad (2)$$

where $g_{LR_i}(m_i, w, k)$ calculates the estimated value of the i-th metric based on the current m_i, w and change k.

During the scaling algorithm in a later phase, we need to consider the accuracy of the LR models applied above. For this, we calculate the Mean Squared Error (MSE) as the error of given LR model and R-square values as the 'goodness' of the estimation.

MSE measures the deviation between the estimated values (m'_i) and the real values. The calculation of MSE is as follows in our case:

$$\text{MSE} = \frac{1}{n}\sum_{j=1}^{n}(m'_i(j) - m_i(j))^2 \tag{3}$$

where n is the number of data points, j denotes the index of the observed cases during individual measurements, $m'_i(j)$ are the estimated values and $m_i(j)$ are the real values for the i-th metric.

R^2 indicates how well the model can explain the variance in the data. The R^2 value is calculated as follows:

$$R^2 = 1 - \frac{\sum_{j=1}^{n}(m'_i(j) - m_i(j))^2}{\sum_{j=1}^{n}(m_i(j) - \bar{m}_i)^2}. \tag{4}$$

The MSE and R^2 indicators help us judge how well we have managed to calculate the estimated value of a metric with the help of its associated LR model.

Finally, we organise the estimated values of metrics in a vector, simplifying the description of the input to the neural network. Let us denote this vector as $\mathbf{M'}(k)$, where

$$\mathbf{M'}(k) = [g_{LR_1}(w,k), g_{LR_2}(w,k), ..., g_{LR_n}(w,k)]. \tag{5}$$

This vector forms the input to the neural network. Thus, the input to the function f_{NN} is simply $\mathbf{M'}(k)$, and the response time estimation is $RT'(k) = f_{NN}(\mathbf{M'}(k))$.

3.2 Neural Network for Estimating the Response Time

The neural network (NN) is trained to learn the non-linear relation among the observed metrics and the response time. The goal of the neural network is to estimate the response time based on the vector of metrics i.e. it will predict how the system's response time changes in response to new metric values caused by scaling of resources.

$$RT'(k) = f_{NN}(\mathbf{M'}(k)) \tag{6}$$

where $RT'(k)$ is the estimated response time, f_{NN} denotes the neural network that estimates the response time from the vector of estimated metrics $M'(k)$ for a given scaling factor k.

The two-layered feedforward neural network what we used can be represented as follows:

$$RT' = \theta_0 + \sum_{k=1}^{H_1}\theta_k \cdot f\Big(w_{k,0} + \sum_{j=1}^{H_2} w_{k,j} \cdot f\big(w_{j,0} + \sum_{i=1}^{N} w_{j,i} \cdot m_i\big)\Big) \tag{7}$$

In the (7), we let f denote the LeakyReLU activation function, defined as $f(x) = \max(\alpha x, x)$, where α is a small positive constant; in our case, it is 0.1. Here, θ_0 is the bias term for the output neuron, θ_k are the weights from the second hidden layer's neurons to the output neuron, $w_{k,0}$ and $w_{j,0}$ are bias terms for the second and first hidden layer's neurons, respectively, $w_{k,j}$ are the weights from the first hidden layer's neurons to the second hidden layer's neurons, and $w_{j,i}$ are the weights from the input features to the first hidden layer's neurons. The input features are denoted by m_i.

Note that the network can be easily modified to include additional output variables such as estimates of tail response time or other metrics such as resource usage, power consumption, etc. The size of the hidden layer will have to be adjusted accordingly.

3.3 Selection of Optimal Scaling Factor

The final scaling decision is practically a procedure which selects the optimal k value that minimizes the cost function (*C(k)*), determining the number of virtual machines to be added or removed from the system.

Let the target range for response time be $[RT_{min}, RT_{max}]$. The number of virtual machines that can be changed in one step in the system is $k \in [-7, +7]$.

The goal is to find the k value for which $RT(k) \in [RT_{min}, RT_{max}]$ and the degree of change in k compared to the previous state is within the permitted range of $[-7, +7]$. This decision can be formalised as follows:

$$C(k) = \begin{cases} RT_{\max} - RT'(k), & \text{if } RT'(k) \leq RT_{\max} \\ \infty, & \text{otherwise} \end{cases} \tag{8}$$

$$k_{\text{opt}} = \underset{k \in [-7,+7]}{\arg\min}\ C(k) \tag{9}$$

This formulation precisely chooses the k value where the estimated response time is as close as possible to RT_{max}, without exceeding it.

3.4 Scaling Procedure

Once the LRs and NN model are trained scaling policy works by periodically monitoring request rate and resource usage metrics. The monitored metrics are used as input to the trained performance model to estimate response time. Baseline algorithm uses a monitoring interval of ten seconds, over which the metrics are averaged. Autoscaling is invoked when observed response time exceed the QoS target's 90%, which is essentially the upper limit for scaling out, referred to as LIM_{upper}. In the same way scale-in is initiated when response time dropped bellow 60% of the QoS target, referred as LIM_{lower}. It can be formulated as follows:

$$Action = \begin{cases} \text{Scale-out,} & \text{IF } avg(RT) > \text{QoS} - \frac{\text{QoS}}{100} \cdot 10; \\ \text{Scale-in,} & \text{IF } avg(RT) < \text{QoS} - \frac{\text{QoS}}{100} \cdot 60; \\ \text{Do nothing,} & \text{otherwise.} \end{cases} \quad (10)$$

In the baseline solution, resource regulation algorithm only activated when the response time fell outside the upper or lower band of this range. If the response time at a given moment was greater than $\text{LIM}_{\text{upper}}$, the system is under-provisioned, and the algorithm scales out.

To calculate the necessary number of virtual machines to bring the response time back within the specified range, the baseline algorithm first leverages the metrics predictor to predict post-scaling metrics for a proposed scaling action. Then, it uses these post-scaling metrics as input to the performance model to predict the response time after the proposed scaling. This allows to determine minimum scaling required to maintain response times bellow the QoS target by evaluating scaling options k against the number of given virtual machines w. k can be any integer number between the predefined range (for example between -7 and $+7$) allowing for arbitrary provisioning changes. In other words the scaler is able to add or remove multiple nodes (VMs) simultaneously in response to large variations in load.

In the baseline algorithm 8 easily monitorable and extractable metrics are selected for observation (cpu usage, context switch, interruption, tcp kb in/out, tcp packet in/out). These metrics are then supplemented with the number of requests per second, using a total of 9 variables to describe the current state of the system.

3.5 Key Features of the Baseline Algorithm

After an extensive review of the available methods, we identified Wajahat's solution as the most suitable for our needs. This decision was based on several key characteristics that is going to be detailed in the next paragraphs.

Wajahat's solution is not based on predicting future value of the load unlike forecast methods, such as [29]. This is a crucial advantage in environments where metrics like request rates and resource demands are highly variable or unreliable. Instead, the method uses externally observable system metrics (e.g., CPU usage, network traffic, memory usage, response time, etc.) to learn how scaling influences Quality of Service Objectives, such as response time.

Another crucial feature expected for our research is that the baseline algorithm should be able to determine the appropriate number of resources (e.g., virtual machines or containers) for scaling. The selected Wajahat method performs exact calculation of the resources in contrast to other solutions such as [30].

Both machine learning models - linear regression for state estimation and neural network for the output performance - can be potentially applied in continuous or incremental learning from observed data, so this mechanism can adapt to changing environment or circumstances.

Unlike systems trained by data originated from publicly available repositories or from simulator, Wajahat's approach learns directly from actual measured metrics of the target hardware and cloud infrastructure. This ensures that scaling decisions are completely optimised for the service in question.

Beyond the previously detailed advantages, the baseline method is empirically validated, its effectiveness has been demonstrated in various environments. This lays a solid foundation for our decision to adopt this method as a benchmark and baseline for our research. Given these capabilities, Wajahat's solution is a perfect fit to demonstrate the research results presented in this paper.

While Wajahat's method effectively addresses many of the challenges associated with autoscaling, we identified several areas for further improvement, six of which we have explored and will be introduced in the next section.

4 Proposed Orchestration Approach

The baseline scaling algorithm presented in Section 3, has already proven to be an effective solution for the

ML-based dynamic regulation of virtual resources. However, through an in-depth analysis of the algorithm, we identified aspects that opened up further research opportunities and directions. While the baseline algorithm provided a solid foundation for dynamic resource regulation, we have expanded upon it in several ways.

4.1 Metric Selection Based on Dynamicity

During the investigation of the baseline algorithm, the focus shifted to the behaviour of the metrics. Our observation (after empirical studies) on metrics is that universal set of metrics cannot be applied to different applications, since resources influencing response time can vary significantly. For example, in a data-intensive application, disk I/O metrics may be crucial, whereas for a computation-intensive application, CPU utilization might play a larger role.

Not all metrics are equally informative for every application's scaling scenario. Some metrics might remain static or show negligible variation during scaling activities, making them less useful for our purposes. To efficiently filter these out, we adopted a statistical test-based method, the independent t-test, to compare variations in metrics before and after scaling actions. The choice of independent t-test as our analytical tool was influenced by the extensive research in the field of feature selection. As highlighted in the comprehensive survey by [31], numerous established methods exist for addressing similar challenges in feature selection.

First, synthetic workloads were used to stress the system in our experiments. This workload was designed so that the load gradually increased to a certain point then decreased in the same way, and this process was repeated several times. During this stress test we systematically increased and decreased the number of resources to observe and measure the changes in metrics and the system's response time. With the data gathered through these controlled manipulations, we were able to train our models, enabling a data-driven approach to scaling decisions.

In the metric selection process, we conducted thorough analyses using measurement series of varying lengths, specifically with sample sizes of 200, 400, and 720 observations. Although we performed several long measurement series, our analyses showed that the set of metrics selected based on the independent t-test remained consistent across the different lengths of data collection. We included only those metrics in our analysis where the t-test indicated a significance level of less than $p < 0.001$. The t-test may indicate statistical significance for large sample sizes even in cases when the effect size is small. As a consequence, we suggest that the metric selection criteria may be altered by Cohen's d especially in cases where the training sample is large, as the t-test can be significant even when the effect of up- or down-scaling for a given metric is small.

This method ensures that our scaling strategy is precise, tailored to individual application requirements, and grounded in rigorous statistical analysis. It allows us to focus on metrics that genuinely influence the system's response time and overall performance during scaling activities. Metrics exhibiting no substantial differences in response to both scaling up and down are considered irrelevant for that specific application's scaling context and are excluded from further consideration. This approach enhances the efficacy and sophistication of our autoscaling solutions.

We identified two advantages of metric selection applied on the baseline algorithm. First, the scope of metrics used for scaling has been significantly expanded beyond the initial 8 metrics identified by the baseline algorithm presented in [27], increasing our estimations reliability. Second, it became possible to effectively exclude metrics that are irrelevant for the scaling of a specific application, thereby increasing the effectiveness of the algorithm and reducing computational overhead.

4.2 Metric Selection Based on Accuracy

Our investigations revealed that there are metrics for which post-scaling values could not be accurately estimated using a linear regression model. This may cause significant inefficiency in our scaling algorithm. To overcome this inefficiency, we investigated further criteria for metric selection.

If we estimate a metric's post-scaling value using (1), then this estimate must be sufficiently accurate for us to use in regulation. Sufficient accuracy is determined by comparing the actual observed m_i' value with the model's estimated value, and in case the model's fit exceeds our predetermined acceptable level, we continue using that metric in future estimates; otherwise, it is excluded from further use. This predetermined acceptance level is defined here by the proportion of variance explained by the model, or the R^2 value calculated in (4). We used the $R^2 > 0.9$ acceptance threshold to keep a particular metric.

While during metric selection based on dynamicity in Section 4.1, we only examined whether the value of a given metric changes as a result of scaling, in the next step the metric has been investigated to decide whether it can be used for estimation i.e. to estimate a metric's post-scaling value based on its pre-scaling value and scaling.

We demonstrate that the range of relevant metrics varies application by application, necessitating automated selection of appropriate metrics when training the regulator using metric selection based on dynamicity and accuracy. Otherwise, the neural network might 'perfectly' estimate response times, even on unseen test data, but we would not be able to effectively regulate the system with it.

4.3 Periodic Evaluation

The main goal of a scaling algorithm is to keep one of the target metrics within a predefined range. In the aforementioned situation, the target metric was the response time in case of the web service as an example. In the baseline algorithm, the calculation to decide on a possible scaling action was initiated when the value of response time has left the predefined range. It is evident that a scaling action therefore was only initiated after the target metric exceeds its minimum or maximum value and the service can only recover after a certain time when scaling happened. The aforementioned operation is considered a reactive scaling.

The next proposed improvements is to make the initiation of decision on scaling independent of external signals and metrics. The scaling logic should independently decide on the necessity of scaling at regular intervals, based on the measured metrics, without reacting to QoS violations happened already. The advantage of this improvement is to let the control loop start a scaling action significantly before the QoS violation happens i.e. when the estimated value of metrics predicts QoS violation in the near future.

Therefore, the scaling procedure consists of periodic evaluation of the observed metrics together with the predicted response time (utilising the LR component). In case the response time is estimated to leave the predefined range, the required scaling action can be calculated (utilising the NN component). This periodic evaluation results a proactive scaling method which is able to foresee and handle the threshold violation of the target metric.

As a result, the proposed solution can respond more swiftly to changing loads. Additionally, scaling algorithms tied to external rules often result in slower reactions, as the triggering signals need to appear consecutively in multiple measurements before scaling is initiated. This is especially problematic for applications where such delays can accumulate, exponentially increasing response times.

4.4 Online Learning

The underlying NN an LR models can be easily retrained to adapt to changes in the workload. This can be advantageous for workloads that are continuously changing in real time (e.g., newer versions of an existing web page, service, REST API, or backend). To make the scaler adaptive, the training of the models are periodically repeated on the dataset extended with the latest observations.

The Neural Network is retrained with one epoch using the observed past data in one batch with a given learning rate with a configurable number of observed events. However, in situations when the type of load generated by the underlying system suddenly changes from e.g. cpu intensive to data intensive, a more sophisticated method is needed. A method for handling this situation (for example re-execution of the metric filtering phase or retraining the NN) is out of scope of this paper.

However, during our experiment when we slightly and slowly changed the load by increasing and decreasing the computational complexity of the given service, the above-mentioned procedure successfully adapted the scaler to the changes.

For retraining the LR models, we had to apply a different approach. When the error in estimation of LR models occurred for example predicted after scaling metrics values deviates from the observed after-scaling metrics values significantly then we adopted an incremental update strategy, applying the stochastic gradient descent method to adjust our LR model's coefficients in an incremental and online way. Which means that we updated the the previous model coefficients c_0, c_1, c_2 as described in (1) with a certain learning rate factor of 0.01. We found that, with the help of this extension, our scaler was able to adapt to subtle and slight changes.

Altogether, by applying the online learning strategy, the scaling algorithm is able to adapt to the slowly changing type of load generated by the observed service and the adaptability has been considered as a significant step ahead in our research.

4.5 Enhancing the Neural Network

Several enhancements related to the structure of the Neural Network (NN) of the scaling algorithm have been applied. First, the number of intermediate layers of the NN has been increased from one to two in order to better capture a) the increased number of metrics and b) the nonlinear relationships between the metrics. Furthermore, the activation function has been changed from sigmoid to LeakyReLU. As a consequence, there is no need to normalize the incoming values of the metrics.

Due to the more complex architecture, the NN component of our scaling algorithm can easily become prone to overfitting. To avoid overfitting, we introduced a Dropout layer after the intermediate layers, which is considered justified based on the following study [32].

As a further protection against overfitting, instead of a fixed number of training epoch, we used a specific stopping criterion. During each epoch, we used 90% of the training data for training and 10% for validation of the estimates. Training stopped when the error measured on the validation data did not reach a new minimum within 100 consecutive iterations or when the number of iterations reached 500.

In the new setup, the input and output variables of the neural network are not standardized or normalized. We realised that the Adaptive Moment Estimation (Adam) optimizer [33] with an initial learning rate of 0.01, led to a smooth and steady decrease of the error during training.

4.6 Eliminating the Request Rate

Many studies dealing with automatic scaling of virtual resources focus on the number of incoming requests in case of web services. This approach is commonly used as changes in the number of requests directly affect system load, thereby providing a clear measure to foresee the load and to decide on scaling. The correlation of the number of incoming request and the load generated by the requests are very high, so that in certain situations even a single metric, the request rate, is enough to decide on scaling even without machine learning algorithms. For those web/internet services where requests may generate different load, this approach can be utilised with a significantly limited success.

Decision on scaling based solely on internal metrics, without the knowledge of incoming request rate, is not so straightforward. The proposed improvement deals with this direction. We modified the scaling procedure to exclude the request rate as obligatory metric (but still keep it as optional). We investigated the new mechanism and found that the scaling algorithm is able to optimally adapt to the specified QoS level/requirements.

This approach marks a new direction compared to the baseline algorithm since the scaling logic can scale independently from the number of incoming requests and as a consequence it supports incoming requests generating varying load without a corresponding change in the rate of requests.

Indeed, there are load scenarios where not only the number of requests varies, but also the complexity of the request submitted. This can be particularly crucial in cases where the system's performance depends not just on the quantity of requests, but also on the complexity or computational demands of those requests.

5 Improvements by Metric Selection Techniques

Regarding the baseline algorithm, it relies on eight metrics and the frequency of incoming requests. In contrast, we significantly expanded the number of metrics examined from 8 to 128, to gain a more comprehensive understanding of the state of scaled applications.

These metrics were chosen arbitrarily and generally include CPU (21 metrics), RAM (30 metrics), BUD (memory fragmentation), DSK (9 metrics), INODE (4 metrics), NET (18 metrics), and TCP (32 metrics), with potential for expansion to include application-specific metrics. For complete and detailed documentation of the metrics that has been collected, please see the manual of the collectl[1] software.

The key is not the specific metrics chosen, but selecting a set that will be responsive to load changes and correlate with response times for a particular application. Our decision to focus primarily on hardware resource metrics was motivated by a goal to maintain comparability in our measurements across diverse applications, ensuring a consistent and standardized basis for our analysis.

In our enhanced procedure, we automatically select the set of relevant metrics, thereby increasing the accuracy of our estimations. This improvement in estima-

[1] Ubuntu 22.04 LTS collectl (2023). https://collectl.sourceforge.net

tion accuracy, in turn, enhances the efficiency of our scaling regulation.

We denote the metric value recorded before scaling as m_i, and the value after scaling as m'_i. The difference between these two values is represented by Δm_i

The change in the number of VMs, compared to any preceding point in time t, is denoted by ΔVM_t and is dichotomously coded such that if $\Delta VM_t < 0$, then it is coded as -1; if $\Delta VM_t > 0$, then as $+1$. A value of $\Delta VM_t = 0$ is not considered, as it indicates no change in the number of VMs.

Based on statistical hypothesis [31] we examined whether there is a statistically significant difference in the Δm_i changes for each metric m_i during scaling up and scaling down. To do this, we perform an independent two-sample t-test for each metric separately.

$$\mathbf{H0}: \ avg(\Delta m_i^-) = avg(\Delta m_i^+) \tag{11}$$

The null hypothesis (see (11)) states that there is no significant difference in the means of Δm_i between scaling up and scaling down for each metric in the context of the examined application.

$$\mathbf{H1}: \ avg(\Delta m_i^-) \neq avg(\Delta m_i^+) \tag{12}$$

The alternative hypothesis (see (12)) suggests that there is a significant difference in the means. We conduct this analysis at a 5% significance level, denoted by an alpha value of 0.05, considering only those metrics where the test shows significant results.

If the p-value obtained from the t-test for a specific metric is less than 0.05, we reject the null hypothesis, indicating that there is a statistically significant difference in the Δm_i changes between scaling up and scaling down for that metric. In such cases, we conclude that scaling significantly affects this metric in the context of the examined application. Conversely, if the p-value is greater than or equal to 0.05, we fail to reject the null hypothesis, suggesting that scaling does not significantly impact that metric. This conclusion guides us to exclude metrics with non-significant changes from further consideration in our scaling algorithm.

By setting the significance level at 5%, we maintain a balance between sensitivity to changes and avoidance of false positives. This means we are 95% confident that the metrics deemed significant truly vary with scaling activities. Metrics that do not show significant changes at this threshold are deemed insensitive to scaling and are thus excluded from further consideration in our scaling process. This methodological rigor ensures that our scaling algorithm is based on metrics that are

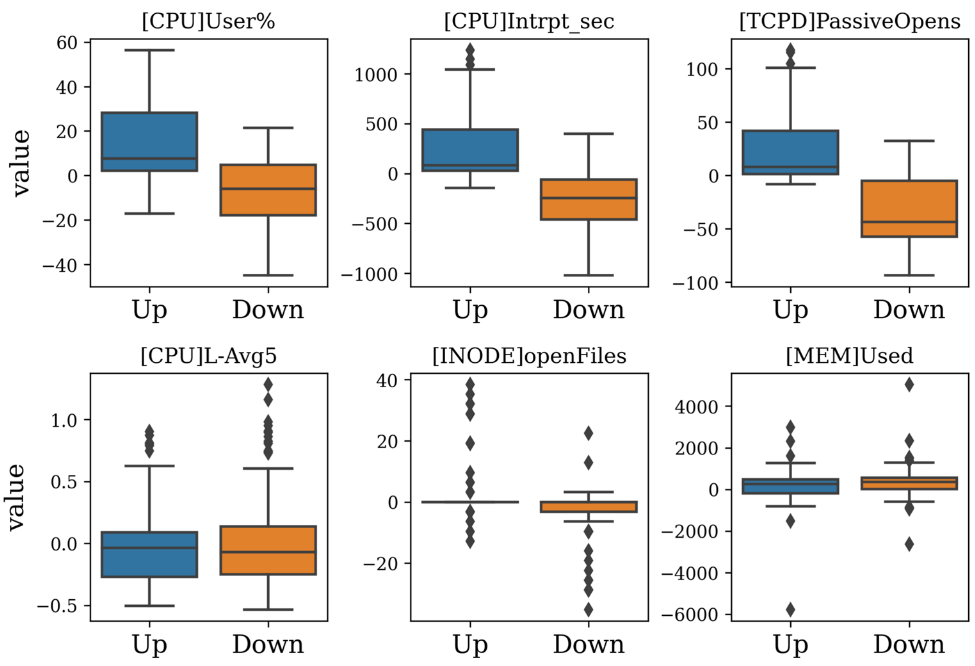

Fig. 1 Variation examples for six different metrics

genuinely responsive to the changes in resource allocation, enhancing the precision and reliability of our scaling decisions.

Figure 1 shows how the values of six chosen metrics changed due to scaling up or down. In examining how the value of a specific metric changes with scaling, the upper row of the diagram reveals significant variations in three metrics - [CPU]User%, [CPU]Interruption per second and [TCPD] passive open sockets - as a result of scaling. Conversely, in the lower row, three metrics - [CPU]L-Avg5, [INODE] open files and [MEM] Used - demonstrate no significant changes in their values regardless of scaling up or down for a particular application.

Three applications (JavaSpring, MongoDB, JavaML) have been selected and investigated to determine the relevance of various metrics in response to scaling. Using the independent t-test, we segregated metrics based on whether their changes due to scaling were statistically significant or not. It showed that the set of pertinent metrics varied across different applications, confirming our hypothesis that application type influences which metrics are affected by scaling. This finding underscores the importance of precise and targeted metric selection for efficient scaling.

Figure 2 illustrates 7 randomly selected metrics with 3 applications. A comparative analysis of the normalized metric value changes for JavaSpring, MongoDB, and JavaML applications are presented. The box plots highlight the median, quartiles, and variability within each metric category. For each metric scaling up (+) and scaling down (-) are shown for the three applications.

The investigation showed that the intensity metrics respond to scaling differs from one application to another. For example, as depicted in Fig. 2, the metric called ”[CPU] interrupts/sec” exhibited change for the JavaSpring application, while no significant change was observed for MongoDB and JavaML. Similarly, the metric ”[CPU] ProcRun” responded to scaling in MongoDB, but not in JavaSpring or JavaML. The aim of this illustration is not to delve into each metric’s detailed analysis, but to highlight the variability in the metrics affected by scaling across different applications.

Principal Component Analysis (PCA) of two load scenarios shown in Fig. 3 has been conducted to examine how the values of 128 metrics changes under the

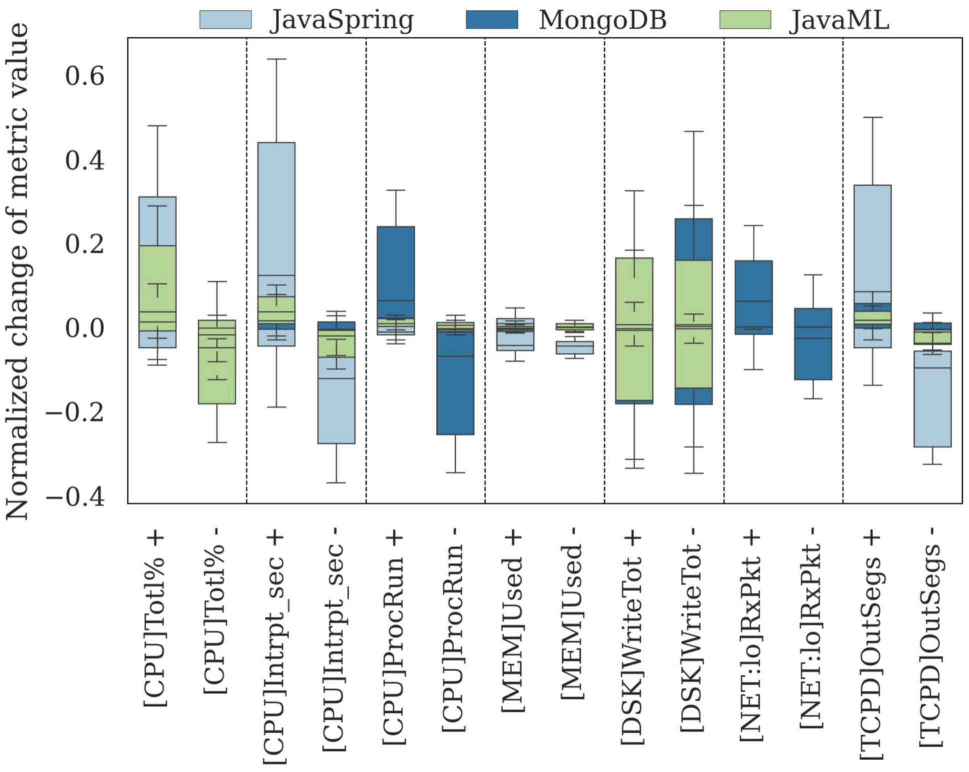

Fig. 2 Normalized metric value changes for 7 metrics

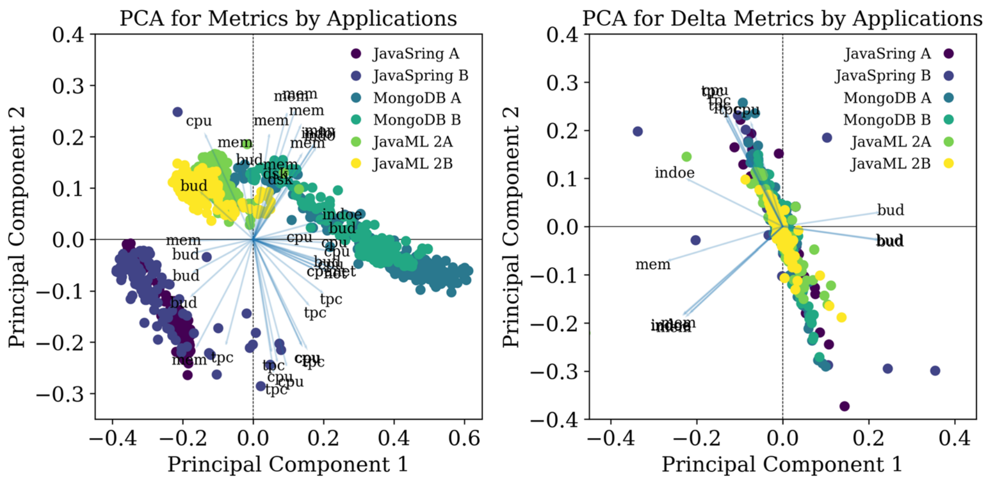

Fig. 3 PCA of 128 metric values under two load scenarios

influence of load in all three applications. Two subspaces in which the metric values were projected do not carry intrinsic meaning but serve as a means to visualize the data.

According to the definition of PCA, it establishes orthogonal dimensions that maximize the variance of the data. Therefore, the first principal component is the one on which the variance of the data is the highest, and the second principal component is orthogonal to the first one. In this subspace, we visualized the metric values associated with different types of applications, observing that the metric values corresponding to each application type cluster distinctly in different locations within this subspace, forming separate clusters.

Based on the results shown on PCA the conclusion is that metrics exhibit unique shifts in response to load for each application, but the directions of these shifts remain consistent irrespective of scaling effects. In other words, among the applications examined, there was no case where the metrics changed positively (i.e., increased in value) in response to scaling for one application and exhibited an opposite direction of change for another application.

Finally, it is essential to note that the selection of metrics was not performed through Principal Component Analysis but rather using the procedure previously described and presented. PCA was utilized solely to illustrate and investigate the relationships among metrics within a lower-, two-dimensional subspace.

Considering metrics that carry only noise and not useful information in system control would significantly impair the effectiveness of the scaling mechanism. This principle stems from the operational logic of the proposed scaling procedure relying on a trained machine learning model. In case the metrics are selected in a way that their values after scaling are inaccurately predicted or not predictable with sufficient precision, feeding these erroneous estimations into a well-trained neural network for response time prediction would inevitably lead to inaccurate and unreliable outcomes. Therefore, a crucial prerequisite of our method is the accurate prediction of post-scaling metric values. This condition, however, is not necessarily met for all metrics and is not self-evident. Our observation indicated that: (1) as previously mentioned, some metrics irrelevant for one application might be relevant for another, and (2) even if a particular metric significantly changes for a given application, its value post-scaling might not be precisely predictable. Consequently, it became essential to further filter the metrics, considering only those whose prediction accuracy reaches a predetermined threshold of $R^2 > 0.9$.

As previously described, the foundation of the baseline algorithm is to estimate the post-scaling value of each metric based on the current metric value (m), the current number of virtual machines (VMs) (w), and the desired change in VM count (k). For every metric measured we compute the m_i' value base on the (1). For each metric calculation, we seek the coefficients $c_{i,0}$, $c_{i,1}$, $c_{i,2}$ associated with the respective metric, which minimize the squared error between the estimated values and the actual data.

While it is possible to find the values of $c_{i,0}$, $c_{i,1}$, $c_{i,2}$ that satisfy this condition in every case, it is not guaranteed that the resulting equation adequately describes the observed data and its variability. Therefore, we use the coefficient of determination, R^2 as defined in (4) to measure how well the estimation of each metric aligns with the actual data. ($R^2 = 1$ indicates a perfect fit of the model to the estimated metric.)

The coefficient of determination, R^2, is a statistical measure that indicates the extent to which our model can explain the variability in the observed data.

The value of R^2 ranges between 0 and 1. A higher R^2 value indicates that the model provides a better explanation for the variability in data. Meaning, if the R^2 for a given metric is high, then the model for that metric is capable of accurately predicting what the values of this metric will be after scaling.

Metrics for which we cannot accurately estimate post-scaling values based on the model fitted to observed data are excluded from further analysis. Otherwise using those metrics could result in imprecise predictions during the control process. Metrics where the R^2 indicator for estimating their post-scaling values does not reach an $R^2 > 0.9$ threshold are not considered.

Since the number of selected metrics 'dynamically' changes according to the specific application, as previously described, the corresponding neural network architecture also changes accordingly based on several considerations.

The number of neurons in the first intermediate layer is set to twice the count of selected metrics plus one. In the second intermediate layer, the number of neurons is determined as half the number of neurons in the first layer, rounded up to the nearest whole number, plus one. The neural network exclusively uses metrics that have successfully passed the previously outlined metric selection process. This results in a variable number of inputs based on the outcome of the preliminary selection.

6 Improvements by Proactivity and Hyperparameter Optimization

6.1 Proactive Scaling

The baseline mechanism only triggers the calculation and decision on scaling when response time exceeds a predefined interval. In contrast, the proposed method conducts continuous (periodic) evaluations to prevent delays. The new approach allows to anticipate QoS violations based on metrics, thus proactively managing the expected deviation of response time from the desired range.

The key is to provide ongoing estimates with the aforementioned predictive models, even for the scenario where the number of resources remains unchanged. This evaluation happens at regular measurement intervals, even if the QoS has not yet been breached and the response time is within the desired limits. If the models are trained accurately and effectively, potential QoS violations can be foreseen based using the function $RT' = f_{nn}((m'_{k=0}))$. This not only ensures more efficient resource utilization but also enhances the overall performance and reliability of the system.

Therefore, the proposed method calculates the RT_{est} value as a function of the M metrics, is capable of "foreseeing" changes in the system's response time. This is why, based on the model's estimate, the algorithm may react before, or as soon as, a QoS violation would occur in the response time as described in (8) and (9).

Thus, the decision to scale is based on these estimated RT_{est} values, rather than any external signal, and depends solely on the metric values and the trained machine learning models. This means that the decision to initiate or stop scaling is based exclusively on the estimations of the machine learning models, and not on the actual measured response time.

The algorithm performing the decision on scaling action is presented by Algorithm 1. It calculates the optimal delta of VMs (k_{op}) required to maintain or improve the system's response time within specified limits.

The algorithm starts by accepting inputs such as the maximum allowable response time (QoS_{max}), the minimum (K_{min}) and maximum (K_{max}) allowable changes in VM count, a vector of metrics (M) representing the actual system resource usage, and the actual number of VMs (w). The output is the optimal number of VMs to scale (k_{op}) between K_{min} and K_{max}, ensuring that the QoS criteria are met or at least as close as possible considering the limitations.

It is important to note that unlike the baseline algorithm, here it is sufficient to define only the upper limit QoS_{max}, and it is not necessary to determine the lower limit at which the scaling algorithm should start scaling downwards.

Algorithm 1 Selecting the Optimal Number of VMs to Scale (k_{op})

```
Require:
    $QoS_{max}$: Upper limit of QoS (RT)
    $K_{min}$: Lowest delta VM (e.g.: -7)
    $K_{max}$: Highest delta VM (e.g.: +7)
    $M$: Vector of Metrics (e.g., CPU usage)
    $w$: Actual number of VMs
Ensure:
    $k_{op}$: Optimal delta VM to meet Quality of Service (QoS)
1: Initialize $k_{op}$ as None
2: for $k \in [K_{min}, K_{max}]$ do
3:     $M' \leftarrow$ empty list
4:     for $i = 1$ to len(M) do
5:         $m'_i \leftarrow c_{i,0} + c_{i,1} \times m_i \times \frac{w}{w+k} + c_{i,2} \times m_i \times \frac{k}{w+k}$
6:         Append $m'_i$ to $M'$
7:     end for
8:     $RT_{est} \leftarrow f_{NN}(M')$
9:     if $RT_{est} < QoS_{max}$ then
10:        $k_{op} \leftarrow k$
11:        break
12:    end if
13: end for
14: if $k_{op} =$ None then
15:     $k_{op} \leftarrow K_{max}$
16: end if
```

Initially (line 1), k_{op} is set to None, indicating that no optimal scaling action has been identified yet. The algorithm then iteratively explores each potential scaling action k within the range [K_{min},K_{min}] (line 2), and calculates the expected system metrics m'_i for each and every metrics (line 4-5) after applying k change in VM count based on (1). The estimated metric value is appended to M' (line 6) which represents the estimating the system's state after scaling by k. The vector M' is then fed into the trained neural network model f_{NN} (line 8), which estimates the response time (RT_{est}) associated with the new metrics values M'.

If the estimated response time for a given scaling action k is below the maximum acceptable limit QoS_{max} (line 9) the algorithm updates k_{op} to the actual k value (line 10) and further computation for other k values is terminated (line 11).

Since we start executing the loop from K_{min} towards K_{max}, this algorithm guarantees that it always selects the smallest possible adjustment to the number of VMs k with which the response time can still be kept below QoS_{max}, thereby optimizing resource use.

In scenarios where no scaling action within the explored range K_{min} and K_{max} can achieve the desired QoS_{max}, the algorithm defaults to (line 14-15) the maximum possible scaling action K_{max}. This decision reflects a conservative strategy to ensure that the system's performance does not degrade below acceptable levels, even if it means utilizing additional resources.

We deliberately choose to initialize the value of k_{op} to be None in a way that provides a means for us to differentiate the case where an optimal k was not found from the ones checked to the case where K_{max} was picked as the optimal just because of algorithmic decision. This difference makes sure that the employment of the maximum resources is a calculated decision. This allows flexibility for other strategies where an optimal k is not identified.

Since the evaluation of scaling is no longer initiated based on external signals, but purely on the estimates made by the models, we introduced the ability for the model to also estimate the response time in cases where the number of resources remains unchanged. Consequently, the model can assess whether the QoS level can be maintained without initiating scaling, based solely on the actual available resources. If the models suggest that the response time meets the QoS requirements, our system will not perform scaling. This innovation ensures that the scaling decision is based solely on the models' estimates. If the models and their estimates are accurate enough, the presented algorithm will not unnecessarily or mistakenly scale the system. However, it may recommend scaling even before the QoS is breached.

When the estimated response time (RT_{est}) is less than the upper limit of the QoS requirements, the scaling is executed, and the specified number (k) of resources are withdrawn from the system. If this estimated value exceeds the QoS upper limit, the algorithm examines the possibility of using one less resource and repeats this process until it finds a solution - a k value - with which the estimated RT falls below the upper limit of QoS. Thus, it always selects the solution that removes or adds as many resources to the system as necessary to bring the RT below the QoS upper limit.

It is important to understand that in many cases actual measured response time is not equal to the response time predicted by the model. As a consequence, the algorithm may suggest scaling even for the situation when actual measured response time is still within the QoS range. The reason for this situation lies in the dynamics of system overload, where changes in metrics become apparent before changes in response time. Response time only becomes visible after task servicing, while metrics are available by real-time monitoring.

This also explains why the actual measured response time of executed requests might still comply with QoS and remain within the desired range, while the model, based on current metrics monitoring the actual system load, predicts that the next measurement cycle will have higher response times exceeding the QoS thresholds.

6.2 Proactive vs Reactive Scaling

The aim of this section is to illustrate the difference between the operation and behaviour of reactive and proactive scaling policies based on the enhanced neural network. In reactive scaling policy, the scaling procedure is invoked when the response time leaves the predefined range. In contrast the proactive scaling policy, the scaler periodically evaluates how many resources is required to keep the response time within the range and can perform changes regardless the actual response time is within or outside the predefined range. During this experiment both scaling policy utilised the same trained model.

When comparing proactive versus reactive scaling policies, it is important to consider that in the case of reactive scaling policies, the setting of the upper and lower threshold values itself is a hyperparameter. In case of proactive scaling policies, we have eliminated the lower threshold value, which is beneficial because we have observed that determining the ideal lower threshold value is not straightforward, and its value impacts the scaling.

Figure 4 introduces the different behaviours of the reactive and proactive scaling policies. There are two graphs above each other, both have the elapsed time on their x-axis synchronised. The upper graph for its y-axis shows the value in time for three measured variables: 1) actual load (request rate, green dashed line), 2) response time in case of reactive scaling (reactive policy RT, blue line) and 3) response time in case of proactive scaling (proactive policy RT, orange line). Moreover, the upper graph also shows the QoS target and lower limit of the response time to be kept by the policies, they are marked with horizontal dashed lines. The lower graph shows the calculated number of virtual machines in time, represented by blue line for reactive and orange line for proactive scaling policy.

Figure 4 clearly shows the dynamic behavior of proactive and reactive scaling policies, with some important observations marked by vertical lines labeled from (a) to (e), which have a correspondence with important moments of the experiment where the proactive policy is far ahead in scaling actions compared to the reactive policy when load increases or decreases.

Before point (a), when load starts to increase (t=7), the proactive policy initiates gradual up-scaling. This very early response is what significantly reduces the number of QoS violations against the reactive policy, which requires that scaling starts only at point (a) (t=10), when response time has already exceeded target QoS. Due to this delay, the reactive policy leads to a larger number of QoS violations and also requires a more extensive up-scaling, as can be noticed by the sharper increase in the VM count.

Similarly points (b) and (d) show the benefits that the proactive policy has in cases of load decline: It down-scales earlier, reducing the number of active VMs

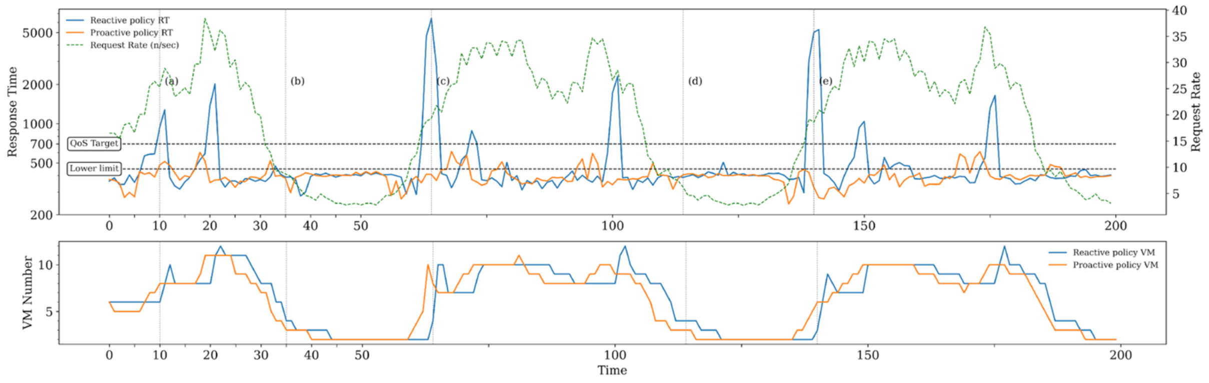

Fig. 4 Comparison of proactive and reactive scaling policies, focusing on response times (RT) and the number of virtual machines (VM) over time

more effectively than the reactive policy. This earlier down-scaling not only minimizes resource usage but also prevents over-provisioning.

As illustrated in the figure, due to these timely settings, this proactive policy has significantly fewer QoS violations. Only 0.72% of all submitted 335,000 requests exceeded the upper limit of the QoS level compared to 5.10% with a reactive policy. Moreover, during this experiment, the proactive strategy used 11% fewer virtual machines.

This is a direct consequence of the proactive policy design that periodically evaluates system metrics and triggers scaling actions before QoS violations occur, rather than reacting after the event.

In our proactive solution, the average number of VMs used was 6.0 with a standard deviation of 3.1, while in the reactive case, it was 6.3 with a standard deviation of 3.3.

The number of QoS violations substantially decreases because the VM count can react to impending workload changes based on metrics, before any violation occurs. It is crucial to understand that this adjustment is not based on the forecasted workload, but rather on internal metrics that monitor the system's internal state.

A slight drawback of the proactive scaling policy is that - by nature - it requires more computational resource, since the model needs to be evaluated not only in cases of QoS violations but periodically. However, based on our measurements, we found that this additional computational cost is negligible, the model evaluation took a few tens of a second in case of $K_{min} = -7$, $K_{max} = +7$.

6.3 Hyperparameter Optimization

As detailed previously, the autoscaling process applies two models executed consecutively. The first set of models estimate the post-scaling values of the metrics (m') as a function of changes in the number of VMs (k), while the second model estimates response time (y) based on these metrics (m'). The first model can be mathematically represented as $g(k) \rightarrow m'$, and the second as $f(m') \rightarrow y$. Together, these models can be expressed as $f(g(k))$, where the result of $g(k)$ becomes the input for $f(m')$.

The composition of the two models, when ($k = 0$), serves to estimate the response time in the scenario where the number of Virtual Machines (VMs) remains unchanged. The process involves using $g(k = 0)$ to predict metric values assuming no change in VM numbers, followed by $f(m')$ estimating response time based on these metrics.

Comparing the estimated RT from $f(g(k))$ with the actual observed RT is possible only when $k = 0$. Because we do not have observed RT values which tells us what would be the RT if k would be other then 0.

For example, what the response time would have been with an addition of +7 VMs to the system, is never known. Therefore, the error of the $f(g(k = 0))$ prediction of our two models' composition can be calculated otherwise, it is not possible.

This particular case when $k = 0$ enables validation of the neural network (NN) and linear regression (LR) models by comparing estimated response times against actual measured response time, thus evaluating our combined model predictions' error.

We use Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) to assess performance of the trained neural network model. MAE computes the mean absolute deviation between expected and observed values in relation to the number of predictions.

MAE and RMSE were measured using K-fold (3-fold) Block Cross Validation as discussed by Barrow at al. [34], where the collected dataset was divided into three equal parts with the original order of the data preserved within each partition. In each iteration, two-thirds of the dataset were used for training, while the remaining one-third was used for validation to calculate the MAE and RMSE. This process was repeated, rotating the validation set, ensuring that each part was used for testing exactly once.

Candidate Neural Network models with different hyperparameter settings were compared to each other and ranked by their RMSE calculated as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f(g(k=0))_i - y_i)^2} \tag{13}$$

The most appropriate hyperparameters of a given Neural Network were not unambiguous. Some hyperparameters led to higher RMSE and resulted in a less useful model, while others led to lower RMSE and better approximations.

During our measurements, we observed that the learning behaviour of our neural network varied with different settings, reflecting a range of performance levels during inference. This observation led us to apply hyperparameter optimization. Reflecting on the precision attained post-training, we always selected the best hyperparameter configurations according to the scaled service or application.

Following this procedure was particularly useful because hyperparameters are settings that do not change during the training process but can significantly influence the learning outcomes.

The entire hyperparameter space is constructed as follows: 1) types of optimization include RMSProp, SGD, Adam, AdaGrad; 2) learning rate ranges between 0.001 and 0.1; 3) activation functions are one of sigmoid, tanh, ReLU or LeakyReLU; 4) dropout rate ranges between 0.1 and 0.5; 5) the batch size is 2^4 - 2^{10}; 6) type of input metric normalization is either MinMax or Standardized and 7) the loss function of the Neural Network itself is either MSE or MAE.

To find appropriate hyperparameters for a given neural network, we applied Random Search [35] in the hyperparameter space, setting the number of possible searches to 20 to efficiently explore the hyperparameter space. The investigated hyperparameter configuration could be trained for no more than 100 epochs, while Early Stopping [36] was applied on the validation dataset with a patience of 50 epochs. This means that the training was stopped when the error (RMSE) did not reach a new lowest point within 50 consecutive epochs. However, this procedure does not guarantee that the best hyperparameter configuration will be found but it offers a reasonable trade-off to identify a good candidate.

This performance evaluation allows us to measure, compare, and rank the generated models, optimizing hyperparameters and producing a reliable model before deploying the scaling procedure in a real-world environment.

We could potentially incorporate so-called exogenous variables into the neural network that are not affected by scaling activities, however they are useful for increasing accuracy in prediction. For example, the time of observation can be an exogenous variable, which may contain information related to day-night effects. Another group of exogenous variables could be the past values of the metric or their calculated moving average. These exogenous variables could improve the goodness of the neural network model. However this could serve as a basis for further investigation.

7 Measurements and Validation

The purpose of this section is to demonstrate the teachability and applicability of the proposed model in situations where the number of incoming requests in time (request rate) is constant while the complexity (generated load) of the requests may change randomly.

The experiment presented in this section, will attempt to validate the hypothesis that the proposed model can learn the necessary number of resources based solely on internal metrics, independently of the frequency of incoming requests, in order to maintain the response time.

7.1 Definitions and Environment

This subsection will contain the main definitions and environment applied during the experiments we are going to introduce.

For better understanding the examination of the hypotheses, we divided the measurements into two fundamental experimental groups.

1. "Contant Request Rate": applies constant request rate over time, while complexity of the requests was altered randomly
2. "Variable Request Rate": applies request rate changing over time (using different load patterns) with constant complexity of the requests

The autoscaling performance can be measured by the time it takes to make a scaling decision and the time required to start additional virtual entities. There are approaches focus on user-level performance metrics, such as the number of QoS violations for the application [37] (Jindal et al., 2017). In terms of application-level metrics, the fraction of autoscaler latency during which QoS requirements are violated is crucial. To measure the performance of the autoscaling solution, we define the following metric. Response time violation is defined as the fraction of the sum of concurrent requests, where the corresponding QoS requirement was violated, divided by the total number of requests.

During the measurements we are going to compare our proposed scaling solution with two basic scaling policies:

1. RT: RT-threshold based scaling policy: add 1 VM when $RT_{actual} > RT_{max}$ and remove 1 VM when

$RT_{actual} > RT_{min}$. This scaling logic was introduced only for the purpose of having a baseline comparison with the other methods.
2. CPU: CPU-threshold based scaling policy: add 1 VM when $CPU_{actual} > CPU_{max}$ and remove 1 VM when $CPU_{actual} < CPU_{min}$. This scaling logic was introduced only for the purpose of having a baseline comparison with the other methods.
3. OPT: proposed scaling solution presented in this paper is signed by the ”OPT” label

During the measurements, for each application and for each load profile (i.e. load characteristic, see later), we have to empirically find the ’best’ or ’optimal’ lower and upper RT/CPU threshold limits. This task was not straightforward as many approaches in the field of autoscaling face significant criticism due to their lack of flexibility, such as the challenge of defining appropriate scaling thresholds and related settings. As previously mentioned, configuring an autoscaler can be challenging, and this complexity is amplified when configurations vary across different scaling applications. Frequently, multiple configuration parameters are contingent on achieving the desired balance between costs and QoS violations, reflecting the degree of conservatism the autoscaler should exhibit.

Altogether, we compare the above mentioned two scaling policies (RT, CPU) with our proposed scaling policy (OPT) to demonstrate under what conditions the proposed scaling algorithm works well, and to identify the limitations of CPU and response time-based scaling policies.

Three applications have been selected for testing the scalability where these applications are different regarding the resource type they are utilising during their operation. The goal is to examine how scaling behaves in applications that are known a priori to be measurable by different metrics. Primarily, we investigate if the metric selection process is operational and effectively selects the metrics based on which we can then regulate the resources running under the application. On the other hand, we wanted to check if the proposed scaling algorithm also functions adequately with different types of applications. Therefore, these three applications allowed to carry out the test measurements without aiming for comprehensiveness. The appplications are as follows:

1. JavaSpring: a JavaSE-based Spring web application. The hosted VM contains Apache TomCat[2] web server with computationally-intensive benchmarks.
2. MongoDB: a NoSQL MongoDB[3] application. The database contains read-only of 2 million collections of randomly generated 1024-length strings. Here, the queries were formed by various but fixed-length regular expressions.
3. JavaML: a JavaSE machine learning application based on the WEKA [38] open-source machine learning library. The application was trained to perform a classification task where incoming data had to be preprocessed, classified, stored, and the results sent back to the requester. In this case the the request could be evaluated independently.

The experiments were conducted in the HUN-REN scientific research cloud [39]. Depending on the types of applications, we used configurations of (1vCPU, 4GB RAM), (2vCPU, 4GB RAM), and (4vCPU, 8GB RAM), supplemented with 500 GB SSD storage. The virtual machines ran on Supermicro SuperServer 1029GQ-TVRT servers, equipped with two Intel® Xeon® Gold 6230R processors, each with 26 cores running at 2.1GHz and a 35.75MB cache (150W). These servers also featured twelve 64GB PC4-23400 2933MHz DDR4 ECC RDIMM memory modules, two 480GB Intel® SSD D3-S4610 Series 2.5” SATA 6.0Gb/s solid-state drives, and four NVIDIA® Tesla™ V100 GPU Computing Accelerators with 32GB HBM2 memory and SXM2 NVLink. The network performance was further enhanced by a Mellanox 100-Gigabit Ethernet Adapter ConnectX-5 EN MCX516A with dual QSFP28 ports and PCIe 3.0 x16.

The environment used for running the experiments contained the following main elements:

- Apache JMeter[4] application is responsible for generating load

[2] Apache Software Foundation. Apache Tomcat, Version 8.0.23. (2023). https://tomcat.apache.org

[3] MongoDB, Inc. MongoDB, Version 5.0. (2023). https://www.mongodb.com

[4] Apache Software Foundation. Apache JMeter, Version 5.4. (2023): https://jmeter.apache.org

- Apache Load Balancer[5] serves as the access point to cloud services. The Load Balancer's role is to route requests (generated by JMeter) to the appropriate virtual machine hosting the application
- SpringBoot[6] application receives the requests from the Apache Load Balancer and serves (executes) the incoming request(s). This component is hosted by the VM performing computational tasks and is also responsible for collecting performance metrics for the Scaler entity
- the Scaler (realising different scaling policies) based on the collected data decides on the number of VMs hosting a particular application to be tested. This is the component representing/realising the scaling policy

The Scaler application is responsible for scaling and regulating resource demands. It functions as a complex resource manager, recording currently connected resources based on their IP addresses or other identifiers. The Scaler monitors which resources are attached to the cluster, to which the Load Balancer can direct the load. The Load Balancer measures response time (RT), essential for determining Quality of Service (QoS), and forwards this data to the Scaler, which utilizes it in model development.

Load generation is executed from an external network using the JMeter software, which generates and sends client requests to the Apache Load Balancer, evenly distributing them among the available virtual machines. Load Balancer is also configured to apply no replicas, no timeout and no retry. This is necessary to observe the operation of the algorithms without other factors influencing the results.

Communication with the applications occurs via HTTP REST API, where the applications received incoming requests through HTTP GET and POST protocols. For observation purposes, each response includes the response data generated by the application, unique identifiers of the requests, the response time, the start time of the request, and the parameters attached to the request. Based on these data, it is possible to perform a detailed analysis and evaluation of the scaling algorithm's performance.

[5] Apache Load Balancer, Version 2.4. (2023). https://httpd.apache.org

[6] Pivotal Software. Spring Boot, Version 2.5. (2023). https://spring.io/projects/spring-boot

7.2 Experimenting with Constant Request Rate

In this section, we introduce an experiment where the goal is to validate our proposed model's ability to control resources effectively, particularly in scenarios where a direct correlation between VM count and response time as well as between request rate and response time was intentionally obscured.

Essentially, we intend to show that the pattern of incoming request (rate) is only one variable based on which the response time of the scaled system can be determined. However, the complexity degree of the submitted requests can also affect the response time. Therefore, systems built solely on the pattern or prediction of incoming requests would not be able to scale the resources appropriately in such a situation.

In this experiment the training datasets contained 250 observations, with each observation in 1-minute time intervals. This means that data collection went on for a period of 4 hours and 10 minutes.

The key factor to investigate the aforementioned situation is that the experiment varies the complexity of the requests and the number of VMs during the training phase of the models. During training, the request rate constant was kept constant continuously to eliminate its influence on system response time. The only factor that can influence the system's response time is the complexity degree of the submitted request in the given experiment. The essence of this experiment is to investigate whether the machine learning models could uncover the relationship between system metrics and response time independently of the request rate and VM count.

For further clarification, here is an explanation. First, there are clear linear or exponential relationships between the complexity of the submitted requests and response time (RT). The processing time for more complex requests is bigger than for simpler requests, thus the response time (RT) of the system is also bigger. However, during the experiment, while we load or perturb the service with the load (to generate some observation and also collect the observed data and the behaviour of the real system), we vary the complexity of the requests as well as the number of Virtual Machines (VM) in such a way that this functional relationship between the VM number and the RT cannot be observable. We achieve this by having the system receive both light and quickly serviceable requests when there are few virtual machines connected to the system and also when there are many virtual machines connected. Similarly, the system receives complex and slowly processable requests when there are few Virtual Machines in the system and also when there are many Virtual Machines connected. Thus, the response time obviously changes due to two factors: (1) the number of Virtual Machines connected to the cluster and (2) the complexity of the submitted requests. However, when we trained the models on the observed and collected data the machine learning models are not informed about the actual value of the complexity of the request.

This means that although the changes in response time depend on the complexity of the requests, we deliberately do not give the machine learning models direct access to this information, so they must use internal metrics to learn the relationship between the internal metrics values and the response time. And also models have to learn how the change of the VM number affect the metric values themselves.

In this experiment, we designed a setup that resulted in a dataset where no direct relationship between Response Time (RT) and the number of VMs exists. However, this does not imply the absence of any relationship between VM count and response time. Otherwise, it would not be possible to control the response time through the number of VMs. The key point is that our experiment was structured in a way that this relationship couldn't be directly identified, observed, or established.

In order to investigate how well our proposed scaling model learns the relationship between the response time and the available internal metrics compared to models relying on response time and/or number of virtual machines, we have trained our scaling model five different ways for comparison. They are as follows:

- Model RR: scaling decision is based on request rate
- Model VM: scaling decision is based on the number of virtual machines
- Model RR+VM: scaling decision is based on the combination of request rate and number of virtual machines
- Model OPT: scaling decision is based on the internal metrics
- Model OPT+RR: scaling decision is based on the internal metrics + request rate

The five different trained models have been tested to investigate how precisely they are able to calculate the response time based on the different parameters. The result is visualised in Fig. 5 by five separate diagrams. The diagrams show the goodness of the models, where x-axis represents the actual response time, while y-axis is for the predicted response time. If the individual points that represent a specific measurement are in the main diagonal from the lower left corner to the upper right corner, this means that the response time estimated by the model and the actually measured response time match, therefore the model is accurate. Based on this, we can tell at a glance which model's estimation is good and which model is not. Moreover, the average error (MSE) estimated by each model and the fraction (R^2) explained by the model from the standard deviation of the dependent variable have also been shown in the diagrams. Based on the second metric, we can say that the closer this value is to 1, the more accurate the model's estimation is.

Our findings are summarised as follows. Model RR indicates that estimating response time based solely on the request rate is ineffective. This is not surprising since the request rate was constant throughout the entire experiment. Model VM shows that relying on the num-

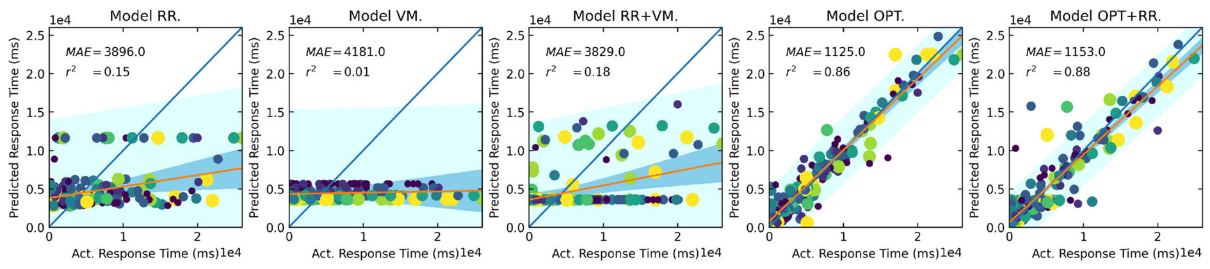

Fig. 5 Scatter plots for five models comparing the observed RT against the predicted RT, each with a 95% prediction and confidence interval. The correlation coefficient (R^2) and Mean Squared Error (MSE) for each model are displayed. The diagonal blue line represents the line of perfect prediction

ber of VMs is even less effective, since we generated training dataset in a way that it contained all the combination of number of VMs and response time. Model RR+VM attempts to predict response time based on a combination of VM number and request rate, which also proves to be ineffective - this aligns with our goal to demonstrate that such a joint estimation is not viable. Model OPT successfully estimates response time using internal metrics, resulting in a good predictive model and finally Model OPT+RR incorporates both internal metrics and request rate in its estimation.

Between Model OPT and Model OPT+RR, we observed minimal improvement, which, however, stems from the phenomenon of the minimal ramp-up time of the requests as they escalated from 0 to 40 req/sec (theoretically, the two Models should completely coincide). Another point is that this discrepancy between the two models, even in this state, is entirely negligible.

Our conclusion is that the addition of request rate does not significantly improve the model's predictive accuracy. However, response time can be quite well estimated based on internal metrics, while estimations based solely on the number of VMs or the incoming request rate (individually or combined) are totally powerless.

We emphasize that this experimental arrangement is only valid for teaching, the teachability of the models, and we only wanted to prove (show) that we can build a scaling algorithm even for cases, when the number of incoming requests or in other words request rate is completely indifferent.

In order to give an insight the way the models were trained in the above experiments, we add additional details in Fig. 6 to illustrate how the Request Rate and number of VM has been made independent from Response Time during the training phase.

The upper left panel in Fig. 6 shows the actual number of VMs during the training. It demonstrates that fluctuations, spikes and dips in response time did not occur merely because of the low or high number of VMs. Hence, there is no clear correlation between the

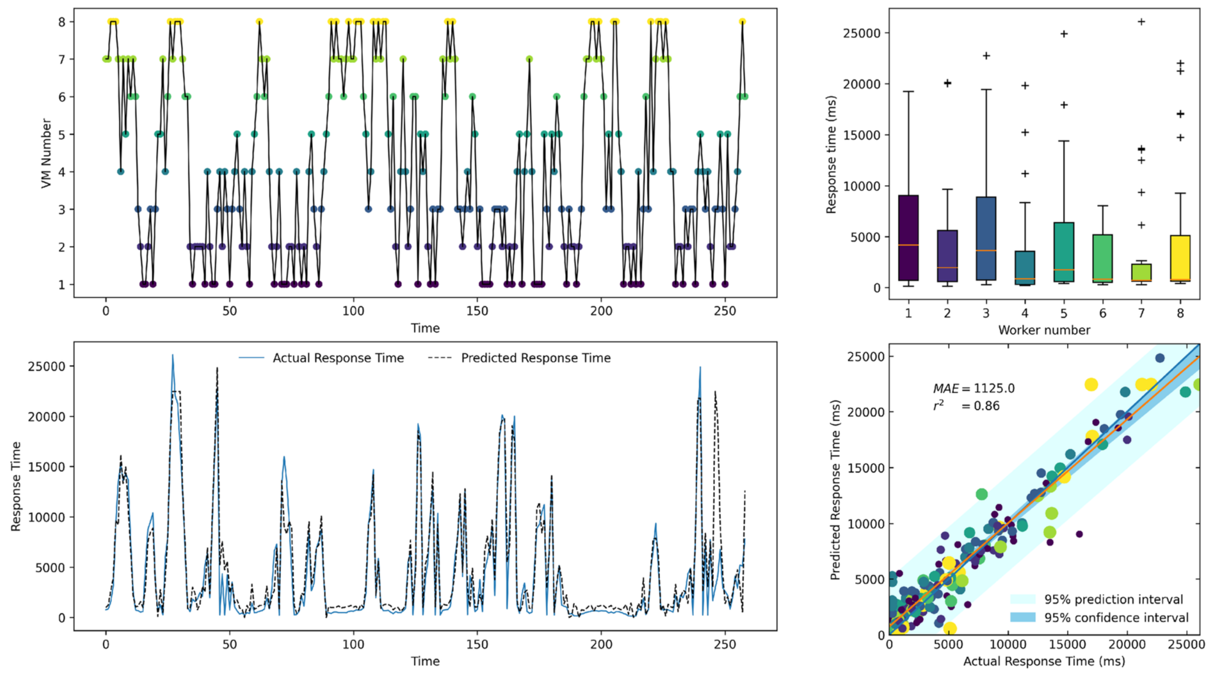

Fig. 6 (Top Left): Observed number of Virtual Machines (VMs) over time. (Bottom Left): The observed ground truth and predicted Response Times (RT) by the models over time. (Top Right): Boxplot of RT grouped by the observed number of VMs, illustrating the distribution of RT across different VM counts. (Bottom Right): Scatter plot comparing the observed ground truth RT against the predicted RT by the model, with a 95% prediction and confidence interval, showing the correlation coefficient (r^2) and Mean Squared Error (MSE)

number of VMs and the RT. The missing correlation was further strengthened by the equal distribution of the number of VMs for all possible Response Time used during the training set. This is shown in the upper right panel of Fig. 6.

This is important because we wanted to eliminate the possibility that the model could learn a relationship between the number of VMs and RT. It is also important to note that the number of VMs does not form an input to our neural network, but it was crucial to produce a training dataset where even the theoretical possibility of a relationship between the number of VMs and RT in the training data is excluded.

The lower right scatter plot shows the relationship between the response time estimated by our model (Model OPT) and the actual measured response time. The horizontal axis represents the actual measured RT value, and the vertical axis represents the corresponding RT value estimated by Model OPT Data points located along the diagonal line indicate accurate estimation.

Finally, the lower left panel of Fig. 6 shows the actual observed RT (with a blue line) alongside the model-estimated RT (with a black dashed line) over time. The horizontal X-axis of the upper left and the lower left panels are synchronized to each other and display corresponding values. The proximity of the two lines indicates that the prediction is accurate, so the training of our proposed scaling model was finally proved to be successful.

7.3 Experimenting with Variable Request Rate

In this section, we introduce an experiment where the goal is to validate our proposed model's ability to control resources effectively in scenarios where incomming request rate is changing over time, while load of each incoming request is constant. This experiment is designed to observe the fluctuations in the number of incoming requests over time and to monitor how our proposed scaling solution compared to other scaling policies (OPT vs RT, CPU described in Section 7.1) adjust the number of virtual machines while concurrently tracking changes in response time.

The first experiment applies a load characteristic from the traffic of a real e-commerce Web Shop for 2 days. An important characteristic of this load is the absence of large and abrupt changes. Our assumption is that the load can be easily followed and that all three scaling policies (CPU, RT, OPT) will be able to scale the resources effectively.

Table 1 presents a comparative analysis of QoS violations and resource utilization across three applications (JavaSpring, MongoDB, and JavaML described in Section 7.1) under varying scaling policies. The data illustrates each policy's effectiveness in maintaining QoS levels, with a lower percentage of violations indicating better performance. It also showcases the average response times during these violations and the total VM usage, providing insights into the efficiency and resource demands of each scaling approach.

Table 1 Comparison of QoS violations and VM usage across 3 applications for webshop load

App type	Policy			
	RT	CPU	OPT baseline	OPT proact
JavaSpring				
QoS	23.93%	0.08%	4.16%	**0.07%**
RT_{avg}	1005.0	601.0	977.7	760.1
VM$_{used}$ (%)	637 (33%)	1353 (70%)	1088 (56%)	915 (47%)
MongoDB				
QoS	28.50%	**0.08%**	2.01%	0.12%
RT_{avg}	1025.2	700.9	1071.9	581.6
VM$_{used}$ (%)	345 (45%)	560 (74%)	485 (64%)	462 (61%)
JavaML				
QoS	6.3%	0.5%	1.3%	**0.05%**
RT_{avg}	1886.8	1947.2	1982.4	2034.7
VM$_{used}$ (%)	350 (49%)	385 (54%)	414 (58%)	414 (58%)

The bold entries are highlighting the winner values in each row

Figure 7 illustrates the dynamic relationship between response time, VM number, and load during the test for the JavaML application only due to space limitations. The top plot shows the response times for each request in the order they were processed, highlighting spikes due to load surges and subsequent stabilization as additional resources are allocated. While the graph suggests more frequent QoS violations under the CPU policy compared to OPT, the Table 1 clarifies that the vast majority of CPU-managed requests were within acceptable response times, with only 0.5% violating QoS, against 0.05% for OPT.

Figure 7 serves to visually support our findings and enhance understanding, complementing the quantitative data provided in the Table 1. The observed ’runaway’ response times and subsequent equalization post resource integration underline the LoadBalancer’s role in managing system efficiency and QoS compliance. It is crucial to note that our conclusions are drawn from a comprehensive analysis of the data, rather than the graphical representation alone.

In our series of tests using load characteristics of a 2 days real e-commerce Web Shop, we observed a consistent pattern across all three applications: response times surged dramatically when additional resources were lately introduced. This spike was a direct consequence of request accumulation on an insufficient number of server units (VMs). This situation arose from our test environment setup where we deliberately configured the components to (1) disable request drop and (2) avoid imposing upper time limits on request servicing. While this may seem an unrealistic experimental setup, we chose it to focus solely on how well the scaling policies worked, without the influence of other system features or technical solutions.

In our experiment, we found that the OPT scaling strategy used slightly more resources than the CPU approach, but it was more effective in keeping QoS violations extremely low at just 0.05%, using 414 VMs compared to 385 VMs in the CPU method.

It’s important to note, though, that for this particular load profile, we didn’t expect to see a significant difference in performance between the CPU and OPT policies. This load profile contained gradual increases and decreases in load, without any sharp spikes. In such a scenario, both methods are likely to perform well. However, the true strength of the OPT policy is more pronounced in situations with irregular and uneven load

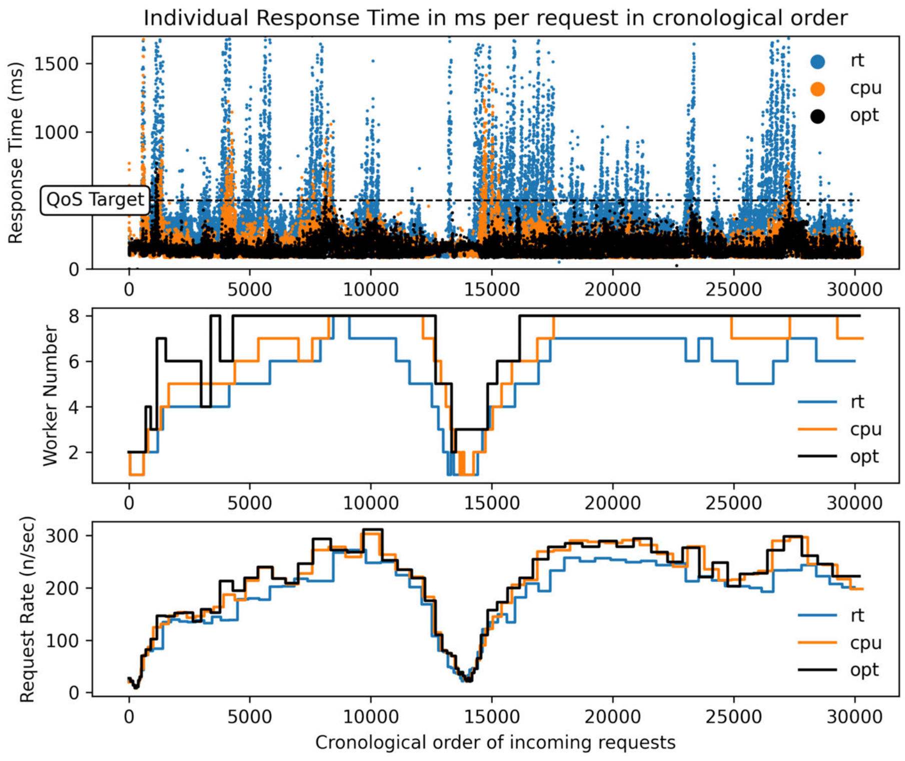

Fig. 7 (Top) Individual Response Time for each request in chronological order, with the QoS target indicated by the dashed line. (Middle) Worker Number reflects the number of VMs over time for RT, CPU and OPT scaling policies. (Bottom) Request Rate over time, demonstrating the load’s nature with step changes

changes, something we clearly observed in our next experiment.

The final experiment is based on an artificial load to explore the effects of sudden and unpredictable surges in load. Our hypothesis is that this load scenario would particularly highlight the advantages of the OPT scaling policy.

We expect the OPT method to outperform the others here because it is capable of quickly determining the required number of resources. This rapid adjustment can prevent request build-ups and, consequently, can avoid increases in response times. We anticipated that under these conditions, the OPT policy's ability to adapt swiftly would be especially beneficial, contrasting sharply with the other two methods.

Before analyzing the values in the Table 2, let's first examine Fig. 8 corresponding to the middle row (MongoDB) of Table 2 to understand the load pattern and where delays in response times occurred.

The key observation of this experiment is the impact of rapid, unanticipated fluctuations in load, particularly evaluating the efficacy of different scaling policies. Our empirical data revealed that under the CPU scaling policy, 13.34% of all concurrent requests encountered QoS violations, in stark contrast to just 1.67% under the OPT policy. This significant reduction in QoS breaches was achieved with a minimal increase in resource allocation by the OPT policy: 1249 VMs (80% of maximum capacity) for CPU and 1142 VMs (73% of maximum) for OPT.

As seen in Table 2, the rate of QoS violation for measuring MongoDB through the OPT Baseline strategy was 8.16% of all the requests. When the OPT Proact approach was used, this rate decreased to just 1.67%. This means that the number of QoS violations is reduced five times, i.e., there is a decrement of 80% compared to the original method.

This outcome was in alignment with our preliminary hypothesis, where we anticipated that the OPT policy, due to its ability to swiftly control the requisite resource allocation, would effectively mitigate request congestion and subsequent escalation in response times. Particularly in the context of MongoDB application, the CPU-based policy's reliance solely on CPU load metrics proved suboptimal. Conversely, the OPT policy, by integrating a broader spectrum of metrics encompassing both CPU and ohter metrics, was able to formulate a more precise correlation between response times, metric values, and VM count. A critical distinction of the OPT policy lies in its capability to make substantial, informed adjustments in resource scaling in a single iteration, potentially adding or withdrawing resources in significant quantities based on its computations.

By inspecting the compiled data table, it is evident that the OPT policy consistently outperformed the CPU-based approach in maintaining QoS compliance, while not necessitating a disproportionate increase in virtual resources. This underscores the OPT policy's superior adaptability and efficiency in dynamic load environments, as evidenced in our meticulous exper-

Table 2 Comparison of QoS violations and VM usage across 3 applications for artificial load

App type	Policy			
	RT	CPU	OPT baseline	OPT proact
JavaSpring				
QoS	10.24%	13.60%	2.95%	**2.39%**
RT_{avg}	2668.7	2838.3	2351.9	2241.1
VM$_{used}$ (%)	289 (18%)	260 (16%)	318 (20%)	301 (19%)
MongoDB				
QoS	28.07%	13.34%	8.16%	**1.67%**
RT_{avg}	2192.6	2909.3	2689.4	1485.0
VM$_{used}$ (%)	788 (50%)	1249 (80%)	942 (59%)	1142 (73%)
JavaML				
QoS	42.40%	11.40%	8.34%	**4.86%**
RT_{avg}	12145.3	9054.9	5887.8	4804.2
VM$_{used}$ (%)	758 (47%)	820 (51%)	806 (50%)	806 (50%)

The bold entries are highlighting the winner values in each row

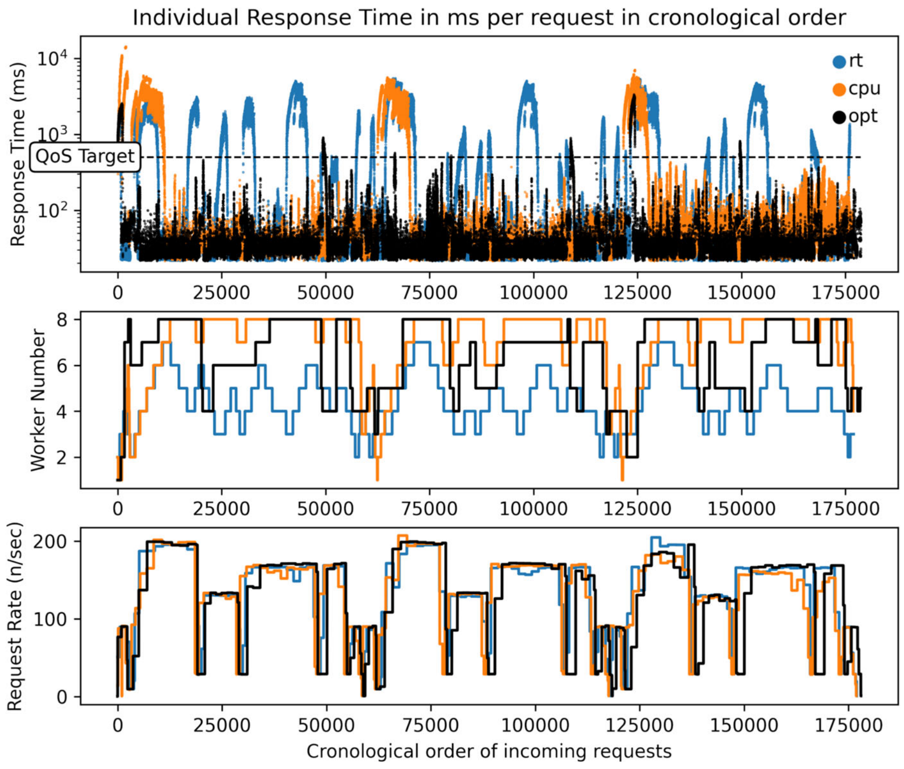

Fig. 8 Response time, number of VM and Request Rate for 3 scaling policies for artificial load. (Top) shows the individual response times per request (Middle) plot tracks the VM count (Bottom) plot displays the request rate over time

imental setup. The most significant improvement has been observed in case of MongoDB experiment where QoS violation from 8.16% to 1.67% resulting in a 79.53% change.

7.4 Advantages and Limitations

One of the main advantages of our method is its ability to adapt to rapidly changing and unpredictable workloads. Unlike traditional autoscaling methods which rely on load forecasting, our system can adjust to variations in load that are not easily predictable. This capability makes our approach particularly effective in environments where workloads can change quickly and unpredictably, maintaining optimal performance and resource utilization.

Another significant benefit is the proactive control mechanism, which anticipates potential Quality of Service (QoS) violations and initiates scaling actions before they occur.

Additionally, the dynamic selection of metrics ensures that the scaling decisions are based on the most relevant data, improving the accuracy and efficiency of the autoscaling process and reducing unnecessary data collection.

Nevertheless, there are limitations. The system learns from observed data, which means it requires an initial period of operation to collect sufficient data to accurately determine the relationships necessary for effective scaling. Careful selection of workload data is crucial during system perturbation. We found that if the workload is not chosen with sufficient attention, a training dataset may be produced in which the learning algorithm could learn incorrect relationships. However, this factor can be avoided if the load is gradually and evenly increased and decreased in the synthetic load. Data collection and perturbation phase should be long enough to gather enough data to understand the workload patterns and resource requirements.

Metrics would be estimated inaccurately could affect the quality of scaling decision as well as overall system efficiency.

The neural network used for the estimation of response times is most prone to overfitting. Hence, it is indispensable to validate trained models against a test dataset for generalizability.

Moreover, the metrics being used while in training may change with any reason in the operational environment, which degrades model accuracy. This can be corrected with periodic comparisons of model predictions against actual data from the operation. If deviations exceed predefined thresholds in these comparisons, the model should be reevaluated and update.

For instance, if the model is based on memory utilization and the underlying infrastructure's memory capacity is increased, then the prediction from the model could be inaccurate. Continuous monitoring with recalibration of the model is necessary to maintain the accuracy of the predictions in such circumstances.

8 Conclusion

This paper presented six design approaches for machine-learning based scaling algorithms which delivered significant improvements compared to the selected baseline algorithm by Wajahat as well as to threshold based scaling algorithms.

A novel metric selection procedure has been presented that identifies metrics relevant for scaling decisions over a particular application. The procedure consists of 1) a Two-sampled t-test filtering to identify metrics with significant effect (Section 4.1) and 2) a linear regression estimation to further identify and exclude metrics with low precision ($r^2 < 0.95$) (Section 4.2). As a result, we were able to raise significantly the accuracy of both the system state models (LR) and the output model (NN) of the baseline algorithm. Measurements (Section 5) have successfully validated the effectiveness of the metric selection procedure.

New approach has been introduced in scaling decision mechanism which enables the calculation of resource needs at any time (Section 4.3) during the lifetime of the application independently from any external events (e.g. QoS violation). Based on this improvement we successfully introduced proactivity (Section 6.1) for the baseline algorithm which enables the controller to intervene before a QoS violation may occur, preventing service degradation. Measurements have been introduced to validate (Section 6.2) these achievements and to underline its importance for burst of loads.

Further improvements related to model accuracy (Section 4.5) has been achieved by optimisation of the hyperparameters (Section 6.3) for the neural network of the scaling algorithm. The optimised hyperparameters have been successfully selected based on the lowest MAE and RMSE to reach the best performance of the neural network.

The enhanced scaling algorithm resulted by applying the three aforementioned improvements on the baseline algorithm has been evaluated by exhaustive and comprehensive measurements (Section 7.3). Series of experiments indicate that QoS violations may reduce by up to 80% (see explanation for Table 2), while the level of resource utilization either remains constant or decreases slightly (by 3-4%) in certain applications. In cases where there was no reduction in QoS violations, the utilization of resources saw a significant decline, falling between 20-50%.

While dependency on request rate is significant among many scaling algorithms in literature, we successfully eliminated this dependency (Section 4.6), so our scaling algorithm is based purely on internal metrics. As a consequence, the algorithm is capable of operating in circumstances where request rate is constant (complexity may still change over time) or irrelevant. Related measurements are described and evaluated in Section 7.2.

Moreover, our achievements will contribute to the European digital research infrastructure developments since the HUN-REN Cloud is a member of the EGI Federated Cloud (a main pillar of the European Open Science Cloud) and the SLICES-RI initiative supported by the European Strategic Forum on Research Infrastructures.

9 Future Work

Our research has provided a strong base for the extension of autoscaling mechanisms in cloud computing by using dynamic metric selection and proactive control. Yet, some open questions remain to further optimize our current approach.

Nonlinear estimation of metrics may help make better predictions in environments where the workload pattern is non-linear by investigating non-linear models such as polynomial regression or neural networks. Further study might be done on how the accuracy of metric estimation affects the efficiency of scaling and how associated uncertainties may be best managed.

In our current solution, uncertainty is handled by increasing the significance level for the t-test to strictly high. However, it worth considering other statistical tests or methods. In the future, we plan to investigate other metric selection methods such as (i) based on the confidence interval of the estimation of a given metric, (ii) implementing robustness check, such as sensitivity analyses to investigate the effects of different selection thresold levels.

One might be interested in estimating post-scaling values of metrics, not based solely on their historical values, but including the values of other metrics and interaction between them. This could yield a better prediction accuracy for individual metrics and hence the overall composite estimation accuracy.

Applying this approach to multiple independently scalable microservices while considering dependencies between them, as done by [40, 41], could also be an area for further investigation, and we would like to pursue research in this direction.

The inclusion of energy consumption indicators during the scale may imply sustainable cloud operation, where energy consumption is kept at the smallest acceptable level, while performance requirements are satisfied at acceptable levels.

Acknowledgements On behalf of the "Autonóm felhöalapú skálázás" ("Autonomous cloud-based scaling") we are grateful for the possibility to use HUN-REN Cloud (see Héder et al. 2022; https://science-cloud.hu) which helped us achieve the results published in this paper.

Author contributions I. Pintye: original idea; design, implementation of algorithms and measurements; writing the draft paper; reviewing relevant research;
J. Kovacs: desiging, editing and writing the paper; support for designing, implementing the algorithms and measurements; revising the work; proofreading;
R. Lovas: conceptualization, validation, proofreading

Funding Information Open access funding provided by HUN-REN Institute for Computer Science and Control. This work was partially funded by the European Commission's Swarmchestrate Horizon Europe project (GA No. 101135012): https://www.swarmchestrate.eu, by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (GA No. F-2.3.1-21-2022-00004) and by the National Research, Development and Innovation Office (NKFIH) under OTKA (GA No. K132838).

Data Availability Statement Data available under private github. The datasets generated and/or analyzed during the current study are available from the corresponding author on reasonable request.

Code Availability Statement Code available under private github. Access is granted upon request.

Declarations

Competing Interests The authors declare no competing interests.

Ethics Approval and Consent to Participate Not applicable.

Consent for Publication Not applicable.

Materials Availability Materials are not publicly available.

References

1. Alipour, H., Liu, Y., Hamou-Lhadj, A.: Analyzing autoscaling issues in cloud environments. (2014)
2. Straesser, M., Grohmann, J., Kistowski, J.V., Eismann, S., Bauer, A., Kounev, S.: Why is it not solved yet?: Challenges for production-ready autoscaling. In: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, (2022). https://doi.org/10.1145/3489525.3511680
3. Zhong, Z., Xu, M., Rodríguez, M., Xu, C.-Z., Buyya, R.: Machine learning-based orchestration of containers: A taxonomy and future directions. ACM Comput. Surv. **54** (2022). https://doi.org/10.1145/3510415
4. Ullah, A., Kiss, T., Kovács, J., Tusa, F., Deslauriers, J., Dagdeviren, H., Arjun, R., Hamzeh, H.: Orchestration in the cloud-to-things compute continuum: taxonomy, survey and future directions. J. Cloud Comput.-Adv. Syst. Appl. **12** (2023). https://doi.org/10.1186/s13677-023-00516-5
5. Biswas, A., Majumdar, S., Nandy, B., El-Haraki, A.: Predictive auto-scaling techniques for clouds subjected to requests with service level agreements. In: 2015 IEEE World Congress on Services, pp. 311–318. (2015). https://doi.org/10.1109/SERVICES.2015.54

6. Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K.K., Zheng, Y., Yan, M., Zhang, X., Liu, A.X.: Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems. In: Proceedings of the 13th Symposium on Cloud Computing. SoCC '22, pp. 16–30. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3542929.3563469
7. Qiu, H., Banerjee, S.S., Jha, S., Kalbarczyk, Z.T., Iyer, R.K.: Firm: an intelligent fine-grained resource management framework for slo-oriented microservices. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. OSDI'20. USENIX Association, USA (2020)
8. Xu, M., Song, C., Wu, H., Gill, S.S., Ye, K., Xu, C.: esdnn: Deep neural network based multivariate workload prediction in cloud computing environments. ACM Trans. Internet Technol. **22**(3) (2022). https://doi.org/10.1145/3524114
9. Imdoukh, M., Ahmad, I., Alfailakawi, M.: Machine learning-based auto-scaling for containerized applications. Neural Comput. Appl. **32**, 9745–9760 (2019). https://doi.org/10.1007/s00521-019-04507-z
10. Toka, L., Dobreff, G., Fodor, B., Sonkoly, B.: Adaptive ai-based auto-scaling for kubernetes. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pp. 599–608. (2020). https://doi.org/10.1109/CCGrid49817.2020.00-33
11. Bansal, S., Kumar, M.: Deep learning-based workload prediction in cloud computing to enhance the performance. In: 2023 Third International Conference on Secure Cyber Computing and Communication (ICSCCC), pp. 635–640. (2023). https://doi.org/10.1109/ICSCCC58608.2023.10176790
12. Yazdanian, P., Sharifian, S.: E2lg: a multiscale ensemble of lstm/gan deep learning architecture for multistep-ahead cloud workload prediction. The J. Supercomput. **77**, 11052–11082 (2021). https://doi.org/10.1007/s11227-021-03723-6
13. Patel, Y.S., Bedi, J.: Mag-d: A multivariate attention network based approach for cloud workload forecasting. Fut. Gener. Comput. Syst. **142**, 376–392 (2023). https://doi.org/10.1016/j.future.2023.01.002
14. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, pp. 217–228. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2950290.2950328
15. Sabuhi, M., Mahmoudi, N., Khazaei, H.: Optimizing the performance of containerized cloud software systems using adaptive pid controllers. ACM Trans. Auton. Adapt. Syst. **15**(3) (2021). https://doi.org/10.1145/3465630
16. Baarzi, A.F., Kesidis, G.: Showar: Right-sizing and efficient scheduling of microservices. In: Proceedings of the ACM Symposium on Cloud Computing. SoCC '21, pp. 427–441. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3472883.3486999
17. Goli, A., Mahmoudi, N., Khazaei, H., Ardakanian, O.: A holistic machine learning-based autoscaling approach for microservice applications. (2021). https://doi.org/10.5220/0010407701900198
18. Abrantes, A., Netto, M.: Using application data for sla-aware auto-scaling in cloud environments. (2015). https://doi.org/10.1109/MASCOTS.2015.15
19. Podolskiy, V., Mayo, M., Koay, A., Gerndt, M., Patros, P.: Maintaining slos of cloud-native applications via self-adaptive resource sharing. In: 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 72–81. (2019). https://doi.org/10.1109/SASO.2019.00018
20. Rossi, F., Cardellini, V., Presti, F.L., Nardelli, M.: Dynamic multi-metric thresholds for scaling applications using reinforcement learning. IEEE Trans Cloud Comput. **11**, 1807–1821 (2023). https://doi.org/10.1109/TCC.2022.3163357
21. Zhang, Y., Hua, W., Zhou, Z., Suh, G.E., Delimitrou, C.: Sinan: Ml-based and qos-aware resource management for cloud microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '21, pp. 167–181. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3445814.3446693
22. Dang-Quang, N.-M., Yoo, M.: An efficient multivariate autoscaling framework using bi-lstm for cloud computing. Appl. Sci. (2022)
23. Jayakumar, V., Arbat, S., Kim, I., Wang, W.: Cloudbruno: A low-overhead online workload prediction framework for cloud computing. In: 2022 IEEE International Conference on Cloud Engineering (IC2E), pp. 188–198. (2022). https://doi.org/10.1109/IC2E55432.2022.00027
24. Pfeifer, A., Brand, H., Lohweg, V.: A comparison of statistical and machine learning approaches for time series forecasting in a demand management scenario. In: 2023 IEEE 21st International Conference on Industrial Informatics (INDIN), pp. 1–6. (2023). https://doi.org/10.1109/INDIN51400.2023.10218206
25. Li, Y., Lin, Y., Wang, Y., Ye, K., Xu, C.: Serverless computing: State-of-the-art, challenges and opportunities. IEEE Trans. Serv. Computing. **16**(2), 1522–1539 (2023). https://doi.org/10.1109/TSC.2022.3166553
26. Hossen, M.R., Islam, M.A., Ahmed, K.: Practical efficient microservice autoscaling with qos assurance. In: Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing. HPDC '22, pp. 240–252. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3502181.3531460
27. Wajahat, M., Karve, A., Kochut, A., Gandhi, A.: Mlscale: A machine learning based application-agnostic autoscaler. Sustain. Comput.: Inf. Syst. **22** (2017). https://doi.org/10.1016/j.suscom.2017.10.003
28. Wajahat, M., Gandhi, A., Karve, A., Kochut, A.: Using machine learning for black-box autoscaling, pp. 1–8 (2016). https://doi.org/10.1109/IGCC.2016.7892598
29. Toka, L., Dobreff, G., Fodor, B., Sonkoly, B.: Machine learning-based scaling management for kubernetes edge clusters. IEEE Trans. Netw. Serv. Manag. **18**, 958–972 (2021). https://doi.org/10.1109/TNSM.2021.3052837
30. Chen, X., Zhu, F., Chen, Z., Min, G., Zheng, X., Rong, C.: Resource allocation for cloud-based software services using prediction-enabled feedback control with reinforce-

ment learning. IEEE Trans. Cloud Comput. **10**(2), 1117–1129 (2022). https://doi.org/10.1109/TCC.2020.2992537

31. Dhal, P., Azad, C.: A comprehensive survey on feature selection in the various fields of machine learning. Appl. Intell. **52** (2021). https://doi.org/10.1007/s10489-021-02550-9
32. Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014). https://doi.org/10.5555/2627435.2670313
33. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. CoRR. **abs/1412.6980** (2014)
34. Barrow, D., Crone, S.: Cross-validation aggregation for combining autoregressive neural network forecasts. Int. J. Forecast. **32**, 1120–1137 (2016). https://doi.org/10.1016/j.ijforecast.2015.12.011
35. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**, 281–305 (2012). https://doi.org/10.5555/2503308.2188395
36. Prechelt, L.: Automatic early stopping using cross validation: quantifying the criteria. Neural Netw.: The Official J. Int. Neural Netw. Soc. **11**(4), 761–767 (1998). https://doi.org/10.1016/S0893-6080(98)00010-0
37. Podolskiy, V., Jindal, A., Gerndt, M.: Multilayered autoscaling performance evaluation: Can virtual machines and containers co-scale? Int. J. Appl. Math. Comput. Sci. **29**, 227–244 (2019). https://doi.org/10.2478/amcs-2019-0017
38. Witten, I.H., Frank, E., Hall, M.A., Pal, C.J.: Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques, 4th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2016)
39. Héder, M., Rigó, E., Medgyesi, D., Lovas, R., Tenczer, S., Török, F., Farkas, A., Emődi, M., Kadlecsik, J., Mező, G., Pintér, Á., Kacsuk, P.: The past, present and future of the ELKH cloud. Inf. Társadalom. **22**(2), 128 (2022). https://doi.org/10.22503/inftars.xxii.2022.2.8
40. Podolskiy, V., Patrou, M., Patros, P., Gerndt, M., Kent, K.B.: The weakest link: Revealing and modeling the architectural patterns of microservice applications. In: Conference of the Centre for Advanced Studies on Collaborative Research (2020). https://doi.org/10.5555/3432601.3432616
41. Luo, S., Huanle, X., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., Xu, C.-Z.: Characterizing microservice dependency and performance: Alibaba trace analysis. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 412–426. (2021). https://doi.org/10.1145/3472883.3487003