Multi-objective Hybrid Autoscaling of Microservices in Kubernetes Clusters

Angelina Horn[1], Hamid Mohammadi Fard[2], and Felix Wolf[2]

[1] Cronn GmbH, Germany
angelina.horn@cronn.de
[2] Technical University of Darmstadt, Germany
{hamid.fard,felix.wolf}@tu-darmstadt.de

Abstract. The cloud community has accepted microservices as the dominant architecture for implementing cloud native applications. To efficiently execute microservice-based applications, application owners need to carefully scale the required resources, considering the dynamic workload of individual microservices. The complexity of resource provisioning for such applications highlights the crucial role of autoscaling mechanisms. Kubernetes, the common orchestration framework for microservice-based applications, mainly proposes a horizontal pod autoscaling (HPA) mechanism, which, however, lacks efficiency. To hinder resource wastage and still achieve the requested average response time of microservices, we propose a multi-objective autoscaling mechanism. Based on machine learning techniques, we introduce a toolchain for hybrid autoscaling of microservices in Kubernetes. Comparing several machine learning techniques and also our in-house performance modeling tool, called Extra-P, we propose the most adequate model for solving the problem. Our extensive evaluation on a real-world benchmark application shows a significant reduction of resource consumption while still meeting the average response time specified by the user, which outperforms the results of common HPA in Kubernetes.

1 Introduction

The microservices architecture provides a highly flexible approach for design and deployment of cloud native applications. In this architecture, the application is decomposed into a collection of loosely coupled services, which interact over light remote interfaces, such as REST APIs.

Containerization is the major technology to deploy the microservices. In both industry and academia, we could observe a change of focus from virtual machine centric to container centric approaches, in the cloud environments [5]. To manage the life cycle of containers in scale, container orchestration frameworks allow the cloud and application providers to define how to select, deploy, monitor, and dynamically control the configuration of containers inside the cluster.

Usually the load of an application is not distributed equally among all microservices. Thus, we need to scale the microservices individually to meet the service level agreement (SLA) and quality of service (QoS). Autoscaling is the solution for efficiently managing microservices, which is very complex for the application developers/owners.

The major autoscaling trend in Kubernetes, the de facto standard for orchestration frameworks, is threshold-based autoscaling such that the users specify a set of fixed conditions for scaling actions. Currently, horizontal pod autoscaling (HPA) is the main scaling approach admitted in the Kubernetes community. Vertical pod autoscaling (VPA) is not widely accepted yet but is becoming more popular. It has been proven that the HPA approach is usually not efficient [1]. Consequently, we need to focus on more fine grained autoscaling approaches.

In this paper, based on machine learning modeling, we propose a hybrid autoscaler such that the resources could be scaled horizontally, vertically or by combination of both approaches. The goal of our autoscaling mechanism is simultaneous optimization of response time and resource usage cost. Based on our scaling mechanism, we proposed, implemented and verified a toolchain for the whole autoscaling MAPE (monitor, analysis, planning and execution) loop [12] in Kubernetes. We compared several performance models in our approach and empirically proposed the most adequate model for this problem. In addition, we compared our approach with the built-in HPA in Kubernetes and observed its competence in real-world problems.

The rest of the paper is organized as follows. In Section 2, we review the related work and mention the difference of our approach with the state-of-the-art. We present our proposed autoscaling mechanism and toolchain in Section 3. We discuss evaluation of the autoscaling approach in Section 4 and finally conclude the paper in Section 5.

2 Related Work

Resource provisioning for cloud native applications could be managed at different levels. Resource providers focus on efficient assignment of resources to microservices [6], while application developers guide the orchestrator to manage resources more efficiently. Our focus in this paper is resource provisioning at the developer level.

To better understand the related work compared to our approach, we use the taxonomy presented in [12]. This taxonomy has been proposed for autoscaling of virtual machines (VMs) in the cloud but it would be mainly applicable for autoscaling of microservices at the application level. In this section, we first briefly explain the necessary concepts, from the taxonomy, and then we deeply review some of the remarkable autoscaling approaches proposed for microservices.

Autoscaling is the process of scaling resources for a service in an automated manner. Autoscaling of microservices allows us to allocate more resources for microservices when they are under a hefty load and retake extra resources when the load decreases. In general, resource scaling can be done horizontally, vertically or in a hybrid approach. In horizontal scaling, we add or remove the number of resource instances assigned to a service (scaling out/in). In vertical scaling, we increase or decrease the capacity of already assigned resource instance to a service (scaling up/down).

We could categorize autoscaling approaches based on the three web *application architecture* as: single tier, multi tier and service-oriented architecture (SOA). Single tier or single service application is the minimum deployable and scalable component size. Microservices are often referred to as single service applications. Applications that consist of more than one service are respectively called multi tier applications. A

Autoscaler	Scaling Methods	Application Architecture	Session Stickness	Scaling Indicators	Resource Estimation	Scaling Timing
[13]	Hybrid	Single tier	Non-sticky	Hybrid	Machine learning	Proactive
[14]	Hybrid	NC	NC	Low-level	Hybrid	Proactive
[7]	Horizontal	NC	NC	High-level	Hybrid	Proactive
[16]	Horizontal	Single tier	Non-sticky	Low-level	Machine learning	Proactive
[1]	Hybrid	Single tier	Non-sticky	Hybrid	Application profiling	Reactive
[4]	Horizontal	Single tier	Non-sticky	High-level	Machine learning	Proactive
[15]	Horizontal	Single tier	Non-sticky	High-level	Machine learning	Proactive
MOHA	Hybrid (plus selective scaling)	SOA	Sticky	Hybrid	Machine learning	Proactive

Table 1: Comparing our approach (MOHA) to the related work (*NC* stands for *not clear from the publication*)

commonly used architecture of this type is a three-tier application comprising three services: a frontend, a backend, and a database layer (although databases are usually considered not dynamically scalable and therefore ignored in autoscaling). Finally, SOA describes applications consisting of several independent services that interact through lightweight APIs and are not necessarily connected sequentially with each other.

Session stickiness is another aspect of autoscaling mechanisms. If the intermediate status of interaction between a client and an application is saved, the session is considered stateful or sticky. Most autoscalers limit the scaling cluster to be stateless and to support stateful sessions usually they transform stateful servers into stateless servers before autoscaling, for instance, by moving the session data out of the web servers and store them either at user side or in a shared Memcached cluster [12].

Scaling indicators are the metrics observed in the monitoring phase and are the basis for the actions of autoscalers. They can be divided into low-level and high-level metrics. Low-level metrics are observed in physical or virtual machine layers, such as CPU and memory utilization and cache miss rate. Metrics collected in the application layer are referred to as high-level metrics, such as request rate and response time.

Different approaches are used for *resource estimation* in autoscaling. The most basic and widely adopted approach is rule-based resource estimation. It is described by using a set of predefined rules, made up of conditions and actions. These rules are established mainly by empirical estimations and are hard-coded. Application profiling is another approach that describes the process of testing the saturation of resources, by applying synthetic or recorded workloads on application. Other methods to estimate resources are analytical modeling, in which a mathematical model is composed based on theory and analysis, and machine learning which dynamically models the resource usage.

The *scaling timing* can be either reactive or proactive. With reactive scaling, the autoscaler will only react to a given situation and try to counteract it. In contrast to this, with proactive approaches, an autoscaler tries to avoid certain situations, such as

exhaustion of resources, by executing scaling actions in advance. Predictions can be based on given data trends and patterns or external data.

In [13], the authors developed a hybrid autoscaler using model-based reinforcement learning, to guarantee the continuity of the performance of the application while simultaneously minimizing resource wastage. The researchers identified the possible long learning phase as a general problem of existing reinforcement learning approaches.

Autopilot [14] is a hybrid autoscaler developed by Google, designed for their internal cloud. They use an orchestration tool called *Borg*, which manages instances of a job consisting of several tasks. The objective of Autopilot is to reduce the slack (difference between the requested and used resources) of jobs while maintaining stability.

A predictive autoscaling approach based on a long short-term memory (LSTM) neural network was proposed in [7] that provides horizontal scaling, using historical time-series data. LSTM neural networks represent a particular type of recurrent neural networks and are very suitable to predict the next sequence in a time-series data.

A horizontal scaling approach was proposed in [16], which is an ARIMA based autoscaling approach using historical time-series data. It estimates the number of pods based on the predicted load represented by CPU usage. The calculation of the estimated resources is based on the Kubernetes HPA approach. Furthermore, the approach combines the ARIMA method with a signal analysis method.

The autoscaler approach in [1] provides hybrid autoscaling possibilities based on automatic application profiling. The authors evaluated the work based on a set of random workloads which cannot be compared to the behavior of real-world applications.

The horizontal autoscaler proposed in [4] uses RNN (Bi-LSTM) model and only predicts the number of requests. This work is evaluated using a dummy web server and has no resource wastage analysis. Another horizontal autoscaling in [15] similarly uses the number of requests for modeling but ignores the importance of CPU and memory usage. This work also uses a dummy web server that mimics a dataset, for evaluation.

Respecting the taxonomy proposed in [12], we categorized the aforementioned related works and positioned our proposed approach in this taxonomy. As shown in Table 1, these autoscaling approaches present a variety in almost each autoscaling category. They provide a mixture of purely horizontal or hybrid (combination of both horizontal and vertical) scaling capabilities and no approach representing a purely vertical autoscaler. The reviewed mechanisms are mostly stateless and need some workaround to cover stateful sessions. Finally, we noticed that none of these works use reference applications in detail. In the cases where an application was described, the application did not represent any real-world reference application but rather a simulated or prototype simple web service. Hence the autoscaling approaches lack real-world comparison, such as connected microservice ensembling.

Compared to the explained autoscaling mechanisms, our proposed approach will be evaluated as service-oriented architecture (SOA) that uses sticky sessions. It uses both high-level and low-level metrics as scaling indicators. The resource needs will be estimated via a machine learning based performance model (see the Section 3). Since the performance model is based on historical data, the scaling timing is classified as proactive. We benefit from multiple criteria decision making theory with the flexibility of tuning weights for particular purposes (e.g. scaling data- or compute-intensive ser-

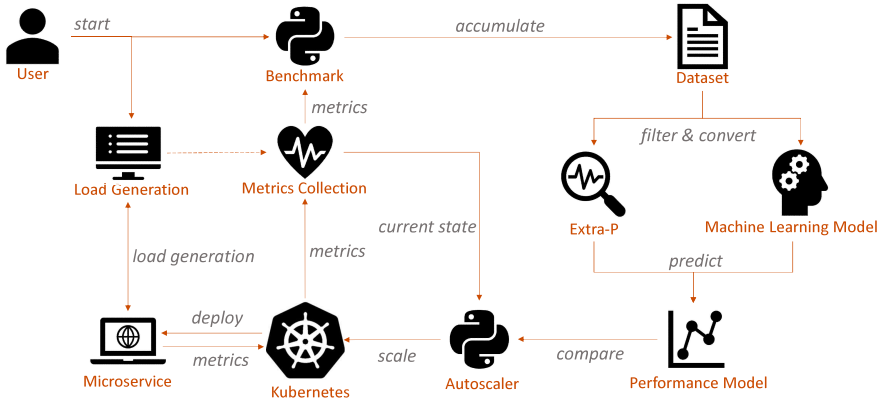


Fig. 1: Architecture of Multi-Objective Hybrid Autoscaling (MOHA)

vices). Finally, our approach provides a flexible hybrid autoscaling such that we could also request for only horizontal or vertical scaling. As analyzed in the Section 4, it will achieve the average response time of services, as the QoS defined by the user.

3 Multi-Objective Hybrid Autoscaling

In this section, we explain our proposed toolchain called Multi-Objective Hybrid Autoscaling (MOHA)[3] and the relation between the components. Figure 1 depicts using MOHA in Kubernetes. We explain the components of MOHA by placing those to three main phases: dataset generation, model training and autoscaling approach. In particular, we discuss the machine learning (ML) models and our proposed autoscaling approach.

3.1 Automatic generation of dataset

The `Benchmark` component calculates a parameter variation matrix based on given input parameters and starts the benchmark for each parameter. The `Load Generation` component applies a given load on the microservice deployed in the Kubernetes cluster with the given resource limits corresponding to the current parameter variation. While running the microservice, the `Metric collection` component collects the various high- and low-level metrics. After the load generation phase, the `Benchmark` module retrieves the collected metrics and filters and summaries these metrics into a dataset.

3.2 Model training

The accumulated dataset of performance data, consisting of the gathered metrics must be preprocessed (by scaling samples, as described in the rest of this section) before the `Machine Learning Model` component can use it. After preprocessing, the resulting

[3] https://github.com/Angi2412/PodAutoscalingKubernetes

data can be converted to be used as a training set for `Machine Learning Model` or any other performance modeling tools.

We model the performance of each microservice based on the current load of the service and the resource limits of pods deploying the microservice. The resource limits include CPU and memory limits, and the number of pod replicas, which are the typical input parameters while defining the pods in Kubernetes. Load of the microservice is dynamically represented by requests per second. The outputs (target metrics) of the model are the average response time, CPU usage and memory usage. The average response time models the performance, while CPU and memory usage model resource utilization. Therefore, we could assume that the utilization also reflects the resource wastage, such that if a given resource is not efficiently utilized, then the resource is wasted.

We chose three different ML models and compared them to explore which one is more suitable for the performance model prediction as part of our autoscaling toolchain. The selected ML models are linear regression (LR), support vector regression (SVR) and multi-layer perceptron regressor neural network (MLPRegressor NN). Since regression models can have multiple inputs but only one output, we trained each model for each of the three target variables. Each of the selected ML approaches represents a different complexity level of machine learning and provides advantages and disadvantages for specific use cases, which inspired us to choose them.

The LR model generally provides fast learning and prediction time while also being suited for large datasets. A disadvantage of the LR model is that it performs well only for linear samples. For this approach, we implemented a least-squares and a Bayesian variant, an extended variant of the maximum likelihood estimator, of linear regression.

The SVR model is more stable against outliers than the other models because of its margin approach. Additionally, it provides a wide range of use cases because of the possibility of using several kernel functions based on the dataset behavior. The advantage over the LR model is handling nonlinear coherence but with much higher training time compared to linear regression (more than quadratic to the number of samples).

The MLPRegressor NN model is a neural network that uses backpropagation for training and the square error as the loss function. MLPRegressor NN can also handle non-linear samples while being especially suited for large datasets with several thousands of samples. Furthermore, it provides extensive customizability by offering several activation functions and solvers. Nevertheless, backpropagation has high time complexity and the training time grows with the number of hidden neurons and layers.

Each ML model owns several hyper parameters, including estimation functions, that need to be tweaked for providing the best possible outcome. Thus, we conducted an extensive grid search for each of the models and hyper parameters. The grid search consists of an estimator, a parameter space, a validation scheme, and a score function. The estimator is the ML model, the parameter space specifies a search space for a hyper parameter, and the validation scheme is used to split a given dataset into several smaller datasets. The number of smaller datasets is dependent on the parameter variation resulting from the number of parameters and size of their parameter space. Finally, the score function is used to compare the accuracy of the estimator given a specific parameter variation. In the grid search, each parameter variation is executed on a smaller dataset and then compared to the performance of the others. The output of the grid search is a

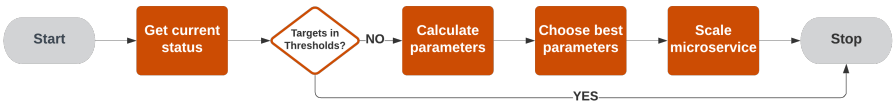


Fig. 2: The autoscaling loop

ranked table of all parameter variations. The described grid search was executed with all three ML models to configure their hyper parameters.

Since we use ML to represent regression then we need to normalize datasets for using distances in the loss functions. We applied the *MinMaxScaler*[4], which scales each sample to a value $1 \leq \alpha \leq 0$. The scaler is fitted onto the training dataset. The inputs and each target use a different scaler instance. These scalers are saved from being usable with new samples. Each time new samples are predicted, the input has to be transformed with the fitted scaler. Furthermore, the predicted values are inversely scaled with the saved target scalers to bring the normalized predicted values into the original ranges.

3.3 Autoscaling loop

The `Autoscaler` component in Figure 1 uses the predicted performance model from the previous phase, to choose the proper parameters for improving its target metrics. It gathers the current status of the microservice, checks if resource scaling is necessary and uses the Kubernetes APIs to scale the microservice accordingly.

Every autoscaler follows the *MAPE* (monitoring, analysis, planning and execution) loop [12]. Monitoring is the first step that fetches all the available metrics for a given application. The second step is to analyze the fetched data regarding specific metrics exceeding a given threshold. Considering the metrics measured in the previous step, we need to plan how to proceed for resource provisioning. In the last step of the MAPE loop, the plan must be executed. The MAPE loop is running in a specified time period, which could be, for instance, the application life cycle. The MAPE loop of our autoscaling mechanism is shown by Figure 2, which is triggered in specific intervals.

First, the Autoscaler gathers the current status of the microservice. It calls the `Prometheus`[5] instances which gathers high-level metrics from the service mesh `Linkerd`[6] and low-level metrics from Kubernetes itself. Then it checks the target metrics. If all targets are in their thresholds, this means that the targets are satisfactory then the loop is exited, and the microservice is not scaled. However, if the threshold is exceeded by any of the three target metrics, the autoscaling loop would proceed. We discussed our precise setting in Section 4.

The optimal resource limits for each target are calculated based on the current target values, parameter status and the aimed values. To calculate the number of pods, we rely on the same calculation used in the Kubernetes HPA [7].

[4] from the Scikit-learn package (https://scikit-learn.org/)
[5] https://prometheus.io
[6] https://linkerd.io
[7] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Each optimal resource limit ($p_i^{optimal}$) for parameter p_i of the web service, is calculated based on the desired target value ($t_i^{desired}$) and the current status ($p_i^{current}$ and $t_i^{current}$), as follows:

$$p_i^{optimal} = \lceil p_i^{current} \cdot \frac{t_i^{current}}{t_i^{desired}} \rceil \tag{1}$$

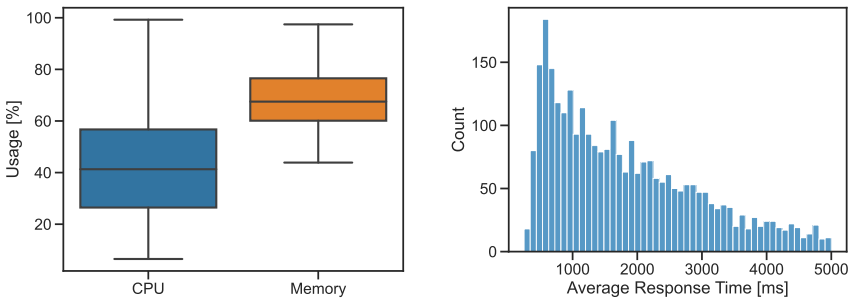
The number of optimal parameters is calculated by *number of parameters* to the power *number of targets*, which in our autoscaler is 27 ($= 3^3$). Based on these calculated parameters, a parameter variation matrix is calculated similarly for generating the synthetic dataset. As a difference, the parameters are not used as ranges but used as discrete values. For calculating the parameter variation, the optimal CPU limit that is calculated based on the memory usage as well as the optimal memory limit that is calculated based on the CPU usage are neglected because of their missing correlation. Considering this removal, our parameter variation matrix includes 12 ($= 3 \times 2 \times 2$) parameter variations. This approach limits the possible decision space of parameter variation, ensuring that only sufficient parameter variations are considered.

Then the parameter matrix is scaled by the same *MinMaxScaler* used to train the machine learning models. Furthermore, for each parameter variation, all three target metrics are predicted by the trained models. The resulting predictions are then scaled inversely to represent their original units.

The array with the predicted targets is then handed over to TOPSIS [9], which is a multiple criteria decision making (MCDM) method. The TOPSIS selects the most suitable target set, considering the weights and criteria. Therefore, the criteria for the TOPSIS are always as minimizing the average response time, maximizing the CPU and memory usage, minimizing the CPU and memory limit, and minimizing the number of pods. Based on what result is desired, the weights of each criterion can be adjusted. Moreover, the output of the MCDM is a ranked list of the targets, with the rank specifying how well a target set fulfils the weighted criteria. Since the ranked list of targets is stated in their original order, it is possible to get the best-ranked target set index and its corresponding parameter variation.

Consequently, the best chosen resource limits are used to scale the microservice with the Kubernetes API. In contrast to the pod update method used in the synthetic dataset generation, the updated pod is not recreated but patched. This approach ensures that the rolling update function of Kubernetes is used. The rolling update functionality ensures no downtime and, therefore, no loss of requests while a pod is scaled [10]. Additionally, a readiness probe is implemented in the deployment YAML file of the scaled microservice. If a pod is added to the deployment in horizontal scaling, the traffic is only redirected to the new pod once it is ready. The readiness is checked by making an HTTP request to a specific rest endpoint of the microservice that responds with the status 200 when it is reachable. In vertical scaling, a new pod with the updated resources is created and checked if it is ready before the previous pod is deleted.

Finally, the autoscaling loop is finished and adds a new run of itself to the Python scheduler with the set scaling time.



(a) CPU and memory usage (b) Frequency of the average response time

Fig. 3: Target metrics

4 Evaluation

We extensively evaluated our proposed approach including the autoscaling loop and the MOHA toolchain. We used the Scikit-learn[8] and Scikit-Criteria[9] packages to implement the machine learning models and the TOPSIS functionalities, respectively. In continue, we first present the experimental setup and then we analyze the achieved results.

4.1 Experimental Setup

We generated a dataset consisting of 3,125 samples with five variations for each parameter in the ranges of *[300m, 500m]*[10] for CPU limit, *[400Mi, 600Mi]*[11] for memory limit and *[1-5]* for replicas. The dataset was split (by randomly shuffling) into the training dataset consists of 75% and the test dataset consists of 25% of the entire dataset.

To be more clear, we depicted a summary of the target metrics of the generated dataset in Figure 3. The boxplots in Figure 3a show that we have the possibility of both upscaling and downscaling, as needed. The frequency histogram in Figure 3b illustrates that the dataset includes various response times, concluding different (light to heavy) loads, were generated.

Each machine learning approach was trained and tested on the same datasets. To ensure comparability between the machine learning algorithms, the used training and test datasets were scaled with the MinMaxScaler. Moreover, for each algorithm, a grid search was performed to tune its hyperparameters.

The threshold for the average response time describes the maximum value that the average response time should not exceed. Respecting the suggestion from [11], this threshold was considered as 1 second. To avoid CPU throttling and out of memory

[8] https://scikit-learn.org/
[9] https://scikit-criteria.readthedocs.io/
[10] m stands for millicore
[11] Mi stands for mebibyte

errors, we should notice that the resources must not be fully utilized. Therefore, the thresholds for CPU and memory usage were set to a minimum of 70% and a maximum of 90%. In contrast to the average response time, the desired CPU and memory usage is the mean value of the minimum and maximum threshold, being 80% in this case.

4.2 The Benchmark Setup

There are several microservices-based benchmark applications, such as robot-shop[12] and the SockShop[13]. But considering the limitation of the applications, e.g. lack of heavy resource usage, we chose the TeaStore [14] to evaluate our approach. The TeaStore is a microservice-based application representing an e-commerce platform, which was intentionally developed for studying microservice behavior [8].

The TeaStore already provides a profile set for real-world user behavior, but this does not include buyer behavior and is implemented as a closed workload model, with no control on arrival rate of service requests. Thus, we reimplemenetd the given user behavior with minor differences to the original profile, which includes randomized buyer behavior. Furthermore, the provided user behavior for the load testing tool JMeter[15] was changed into an open workload behavior that generates a given number of requests per second.

A user after visiting the landing page of the store, performs a login action with a random username. Afterwards, the user visits a random category and product, which the user puts in its cart. This sub loop is repeated randomly up to five times. When all products are in the cart, it is randomly decided if the user buys them or not. Furthermore, the user visits its profile page and finally logs out.

The synthetic dataset was created with a constant number of requests per second, modeling a constant load on the system. The maximum number of requests per second is varied during the dataset generation to generate more diversity in the dataset. Therefore, this load pattern represents only a scenario for upscaling the resource limits. The constant load pattern is suited for the synthetic dataset generation because it provides performance insight from the load on the microservice with a specific resource specification.

We created a custom load pattern based on an adjustable number of requests per second. This dynamic load starts with a constantly increasing load from 1 request per second until reaching a maximum of 1000 requests per second. This increase phase has a duration of three and half minutes. The load of the maximum number of requests per second stays constant for three more minutes. After that, the load is decreased until reaching 1 request per second. Similarly, this decrease happens in a duration of three and half minutes. The profile represents a more realistic load pattern than the constant load pattern used for the dataset generation. It provides periods of increase in requests per second and includes periods of load decrease. Therefore, it is suited for the evaluation runs of the autoscaler by providing up-, down-, in- and out-scaling scenarios and a scenario in which no scaling is necessary.

[12] https://github.com/instana/robot-shop
[13] https://github.com/helidon-sockshop/sockshop
[14] https://github.com/DescartesResearch/TeaStore
[15] https://jmeter.apache.org/

Model	**LR**			**SVR**			**MLPRegressor NN**		
Target	Response time	CPU usage	Memory usage	Response time	CPU usage	Memory usage	Response time	CPU usage	Memory usage
MSE	0.01	0.01	0.00	0.01	0.01	0.00	0.01	0.01	0.00
$\boldsymbol{R^2}$	0.55	0.78	0.90	**0.72**	**0.87**	**0.94**	0.56	0.78	0.90

Table 2: Phase one - Accuracy comparison

Model	**CPU limit [m]**	**Memory limit [Mi]**	**Pods (average)**	**Response time [ms]**	**CPU usage [%]**	**Memory usage [%]**
LR	537.78	**439.85**	2.10	**796.96**	69.32	**78.81**
SVR	**437.47**	460.11	**1.60**	1,344.74	**70.46**	77.41
MLPRegressor NN	583.72	501.44	3.47	1,704.66	66.29	56.65
Extra-P	643.68	578.4	5.25	1,466.55	49.12	52.63

Table 3: Phase two – Qualitative comparison

4.3 Experimental Results

In this section, we analyze the results achieved by our autoscaling toolchain, using the three machine learning models (LR, SVR and MLPRegressor NN) and the Extra-P model [3]. The results are then compared with the Kubernetes HPA's results.

We divided our analysis into three phases, each of them having a different focus. In the first phase, each machine learning approach is trained on the synthetically generated dataset and then their speed of training, speed of prediction and prediction accuracy are compared. In the second phase, we evaluate the results of using ML and Extra-P models in the MOHA toolchain, considering simultaneously minimizing average response time and resource consumption. In the third phase, the horizontal pod autoscaling of SVR is compared to the standard Kubernetes HPA to observe how the performance of our proposed approach outperforms an already established autoscaling approach.

Phase one Since the least squares and the Bayesian variant of the LR delivered precisely the same values, the Bayesian variant is further referred to as LR. To measure the speed of the ML algorithms, we considered an accuracy of 17 decimal digits that were shown by rounding to 4 digits. In our experiments, we observed that the LR algorithm has by far the lowest training time of only 0.7 milliseconds, while the MLPRegressor NN has a much higher training time of 0.2 seconds. Furthermore, the SVR presents the highest training time of all algorithms with around 0.39 seconds. The LR model's prediction time is very short such that after our rounding it could be assumed to be nearly zero. Similar to the training time, the MLPRegressor is the second fastest algorithm with a prediction time of 1.4 milliseconds, while the SVR model has the highest prediction time of 25 milliseconds.

The prediction accuracy was measured with the Mean Squared Error (MSE) and the coefficient of determination (R^2) metrics [2]. The MSE measures the average squared error between the predicted and the actual value. It is always a positive value. The closer the value is to zero, the better the estimator. The coefficient of determination provides

a measure of the probability that the model predicts an unknown sample. The R^2 score can be between infinite negative and one. Here the best value is one, while zero would indicate an estimator that always predicts the expected value regardless of the input. The results of the comparison are shown in Table 2. The best metric value of a target is highlighted.

In summary, it can be concluded that the LR algorithm is the fastest algorithm of all, regarding the training and prediction time but it is the least accurate one. Moreover, the MLPRegressor NN is slower than the LR but slightly more accurate. Furthermore, it scores the same MSE values as the LR and SVR. Finally, the SVR is much slower than the other two methods but results in a much better R^2 score throughout every target. It is, therefore, the most accurate machine learning method of the methods compared. However, considering the importance of prediction accuracy and the fact that the scaling decision is not made every second, the higher prediction time of the SVR could be practically neglected.

Phase two We compare the ML and Extra-P models, embedded into the MOHA toolchain. The comparison is divided into qualitative and quantitative comparisons.

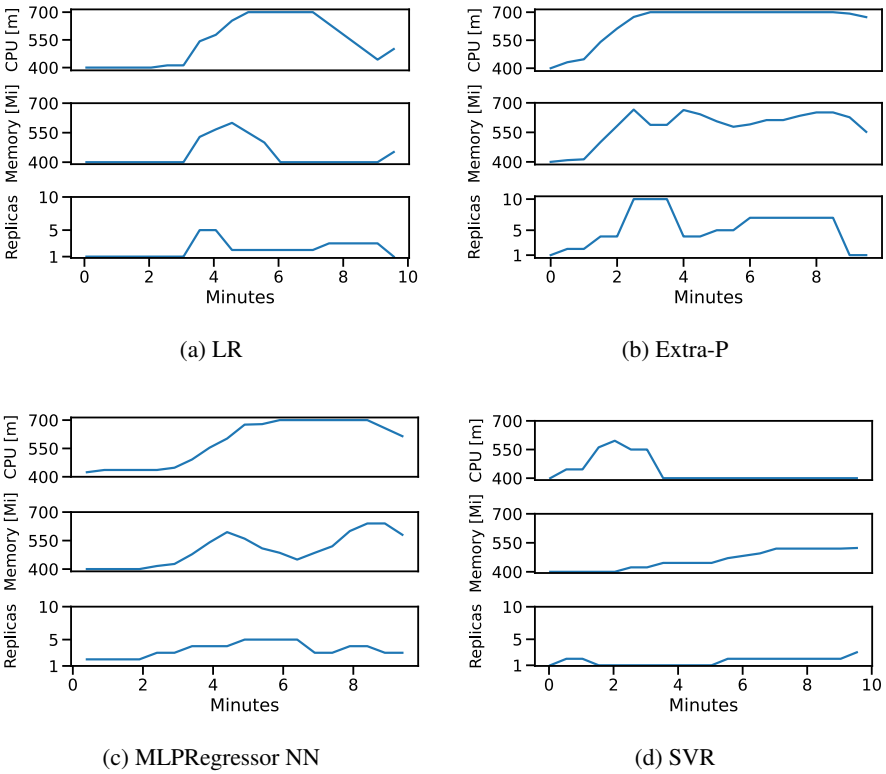
Qualitative comparison We calculated the average of parameters and targets. The Extra-P resource estimation equations were created using its multi-parameter model that uses the median and the refined approach of the current version of the application.

The results of the comparison are summarized in Table 3. Here, all parameters and target values represent the average except for the response time represented by its median. The median response time is chosen to get a more outlier free impression of the behavior. Moreover, the TOPSIS decision maker's weights are chosen to be balanced: the average response time is weighted with 0.5 while all resources related weights sum up to 0.5. The scaling time for each approach is set to 1 minute, allowing up to ten scales per evaluation run.

We observed that the LR model achieves the lowest median response time despite scoring the least accuracy in the prediction of the average response time. Additionally, it scores the highest average memory usage. Furthermore, the prediction with the MLPRegressor NN scores the highest and, therefore, worst response time. The SVR-based autoscaler does use the lowest average CPU limit and number of pods while also showing the highest average CPU usage. It is especially noticeable that Extra-P is scoring the worst values in each metric except for the median response time. This result can be caused by the fact that Extra-P was not initially designed to make precise predictions but rather to find scaling bugs in the performance behavior of a system. Finally, the MLPRegressor NN shows in no metric, except for the median response time, the best or worst value. It can therefore be considered average.

The scaling behavior of all models are shown in the Figure 4. For each approach, a visualization of each parameter was conducted over the time of the evaluation run (in minutes). As shown, SVR uses considerably less resources, considering the number and the resource capacity of pods, while meeting the specified average response time.

Quantitative comparison Since the microservice-based applications usually run on public cloud environments, analyzing the monetary cost of running applications on



(a) LR

(b) Extra-P

(c) MLPRegressor NN

(d) SVR

Fig. 4: Scaling behavior

commercial public clouds makes perfect sense. In continue, based on the published resource prices from the Google Cloud [16], we estimated and compared the cost of running the benchmark application using different autoscaling approaches. To estimate the cost of resources, we assume that the Google Kubernetes server is located in Frankfurt, Germany (Europe-west3). Moreover, the resource prices per hour and a total utilization of 24 hours per day are assumed. Consequently, one vCPU (1000m) costs 0.0573$ per hour, while one gigabyte of memory ($\approx$ 953Mi) costs 0.0063421$ per hour. To calculate each of the resource costs per hour, the average resource limits of the evaluation run are converted to the corresponding units, multiplied by the corresponding prices, and extrapolated to one hour. Finally, the resource costs are summarized and extrapolated to 24 hours to result in the total resource costs per day in the US dollar.

The results of the cost estimation are shown in Table 4. The SVR-based approach shows the lowest costs of CPU and memory resources and therefore, it is the approach with the lowest total resource costs of 6.48$ per day. The Extra-P approach results in a high total resource cost of 30.72$, as the most expensive approach among all.

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

Model	CPU [$]	Memory [$]	Total [$]
LR	9.36	0.96	10.32
SVR	**5.76**	**0.72**	**6.48**
MLPRegressor NN	16.8	1.68	18.48
Extra-P	27.84	2.88	30.72

Table 4: Phase two – Quantitative comparison (cost per day)

Model	$\mathbf{W}_{response\ time}$	$\mathbf{W}_{resources}$
SVR-HPA$_t$	0.9	0.1
SVR-HPA$_r$	0.1	0.9
SVR-HPA$_b$	0.5	0.5

Table 5: TOPSIS weight variations

In summary, we could conclude that the SVR-based approach provides the fewest number of pods, highest resource usage and therefore, lowest resource cost. Moreover, although it scores a higher median response time than the LR, its response time still meets the requested QoS. The Extra-P based model scored worst in the described approach because of its very high resource costs and insufficient median response time.

Phase three We compared the horizontal scaling ability of the SVR-based approach with the Kubernetes HPA. For the sake of compatibility, we implemented the Kubernetes HPA approach in Python. The HPA follows the same process similar to the other approaches for gathering the current status and uses the resource estimation of the Kubernetes HPA. The HPA estimates the required number of pods based on the current number of pods and the used target metric. It then decides to take the maximum number of pods calculated from all target metrics. In the case of the novel implemented autoscaler approach, the calculated CPU and memory limits are neglected in the resource estimation process to only use the horizontal scaling capabilities of the approach.

Table 6 presents the results of the comparison. We set the CPU limit to 300m and memory limit to 400Mi. The maximum number of requests per second was set to 1000, in 1 minute as scaling time, with the maximum number of pods limited to 10.

To study the impact of criteria's weights on the results, several weight distributions of the TOPSIS, following different goals, are compared (see Table 5). SVR-HPA$_t$ favors minimizing the response time, while SVR-HPA$_r$ tends to minimize resource wastage instead. For comparison reasons, SVR-HPA$_b$ representing the balanced weight distribution from phase one is also considered.

It can be observed that the SVR-HPA$_r$ and the Kubernetes HPA use the lowest number of pods. The response time optimized approach uses the most pods, while the balanced approach uses only slightly more pods than the resource optimized and the Kubernetes HPA approach. Furthermore, the resource optimized approach has the lowest median response time.

Overall, the balanced SVR-based approach scores the lowest cost total value and is, therefore, the most beneficial approach of the compared approaches considering the balance of minimizing the response time violations and the total resource wastage.

Model	Pods	Response time [ms]	CPU usage [%]	Memory usage [%]	Total cost [$]
SVR-HPA$_t$	5.55	1141.75	41.36	66.03	8.75
SVR-HPA$_r$	**1.95**	**170**	29.42	**70.09**	3.63
SVR-HPA$_b$	2.15	455.2	**47.35**	62.01	**3.11**
Kubernetes HPA	**1.95**	214.93	34.49	68.56	3.40

Table 6: Phase three - SVR-based HPA vs. the Kubernetes HPA

We could especially distinguish that SVR-HPA$_r$ shows the best median response time even though the intentional weight distribution was not optimized. Furthermore, it shows less CPU usage but more memory usage than the other approaches. In contrast to the resource optimized approach, the response time optimized approach results in the highest median response time of all approaches. This behaviour could be explained by the high number of pods used. It could be possible that the registry web service, responsible for distribution of requests, creates extra overhead for distributing workload to more pods and causes increase of median response time.

All in all, it can be concluded that the SVR-based autoscaling approach shows better results in horizontal pod autoscaling than the Kubernetes HPA when the weight distribution is chosen correctly. Furthermore, the developed approach allows the possibility to tweak its behavior by adjusting its weight distribution to its requirements.

5 Conclusion

Uncontrolled deployment of microservices can cause resource wastage or degrade the performance of applications. In this paper, we propose an autoscaling approach and a toolchain, called MOHA, designed for hybrid autoscaling of microservices in Kubernetes that scale pods horizontally, vertically or in a hybrid way aiming simultaneously decreasing the response time and the resource consumption. We evaluated our approach for three machine learning approaches, namely linear regression (LR), support vector regression (SVR) and the multi-layer perceptron regressor neural network (MLPRegressor NN), and also a performance modeling tool, called Extra-P. We observed that SVR operates better for our problem. Conducting the variant workloads for our experiments on a benchmark application, called TeaStore, confirms the scalability of our proposed toolchain. Moreover, we observed that the horizontal scaling capabilities of the SVR-based approach competes very well with the default Kubernetes HPA for the cost of resource usage while our approach provides much better response time for microservices. We believe that our open source toolchain can be practically used for real-word microservice-based applications and by far decreases the resource usage costs while meeting the QoS for average response time.

Acknowledgements: We acknowledge the support of the European Commission and the German Federal Ministry of Education and Research (BMBF) under the EuroHPC Programme ADMIRE (GA No. 956748, BMBF funding No. 16HPC006K). The EuroHPC Joint Undertaking (JU) receives support from the European Union's Horizon

2020 research and innovation programme and GER, FRA, ESP, ITA, POL and SWE. This research was also supported by the EBRAINS research infrastructure, funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific GA No. 945539 (Human Brain Project SGA3), and is partly funded by the Federal Ministry of Education and Research (BMBF) and the state of Hesse as part of the NHR Program.

References

1. Balla, D., Simon, C., Maliosz, M.: Adaptive scaling of kubernetes pods. In: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. pp. 1–5 (2020)
2. Botchkarev, A.: A new typology design of performance metrics to measure errors in machine learning regression algorithms. Interdisciplinary Journal of Information, Knowledge, and Management **14**, 45–76 (2019)
3. Calotoiu, A.: Automatic Empirical Performance Modeling of Parallel Programs. Ph.D. thesis, Technische Universität Darmstadt (2018)
4. Dang-Quang, N.M., Yoo, M.: Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes. Applied Sciences **11**(9) (2021)
5. Fard, H.M., Prodan, R., Wolf, F.: A container-driven approach for resource provisioning in edge-fog cloud. In: Algorithmic Aspects of Cloud Computing. pp. 59–76. Springer International Publishing (2020)
6. Fard, H.M., Prodan, R., Wolf, F.: Dynamic multi-objective scheduling of microservices in the cloud. In: IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). pp. 386–393 (2020)
7. Imdoukh, M., Ahmad, I., Alfailakawi, M.G.: Machine learning-based auto-scaling for containerized applications. Neural Computing and Applications pp. 1–16 (2019)
8. von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S.: Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). pp. 223–236 (2018)
9. Kolios, A., Mytilinou, V., Lozano-Minguez, E., Salonitis, K.: A comparative study of multiple-criteria decision-making methods under stochastic inputs. Energies **9**(7) (2016)
10. Midigudla, D.: Performance analysis of the impact of vertical scaling on application containerized with Docker, Kubernetes on Amazon web services EC2 (2019)
11. Nielsen, J.: Usability engineering. Morgan Kaufmann (1994)
12. Qu, C., Calheiros, R.N., Buyya, R.: Auto-scaling web applications in clouds: A taxonomy and survey. ACM Comput. Surv. **51**(4), 1–33 (2018)
13. Rossi, F., Nardelli, M., Cardellini, V.: Horizontal and vertical scaling of container-based applications using reinforcement learning. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). pp. 329–338 (2019)
14. Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., Wilkes, J.: Autopilot: Workload autoscaling at google. pp. 1–16. EuroSys '20, Association for Computing Machinery, New York, NY, USA (2020)
15. Toka, L., Dobreff, G., Fodor, B., Sonkoly, B.: Machine learning-based scaling management for kubernetes edge clusters. IEEE Transactions on Network and Service Management **18**(1), 958–972 (2021)
16. Zhao, A., Huang, Q., Huang, Y., Zou, L., Chen, Z., Song, J.: Research on resource prediction model based on Kubernetes container auto-scaling technology. In: IOP Conference Series: Materials Science and Engineering. vol. 569, pp. 1–8. IOP Publishing (2019)