Machine Learning-Based Scaling Management for Kubernetes Edge Clusters

László Toka, *Member, IEEE*, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly

***Abstract*—Kubernetes, the container orchestrator for cloud-deployed applications, offers automatic scaling for the application provider in order to meet the ever-changing intensity of processing demand. This auto-scaling feature can be customized with a parameter set, but those management parameters are static while incoming Web request dynamics often change, not to mention the fact that scaling decisions are inherently reactive, instead of being proactive. We set the ultimate goal of making cloud-based applications' management easier and more effective. We propose a Kubernetes scaling engine that makes the auto-scaling decisions apt for handling the actual variability of incoming requests. In this engine various machine learning forecast methods compete with each other via a short-term evaluation loop in order to always give the lead to the method that suits best the actual request dynamics. We also introduce a compact management parameter for the cloud-tenant application provider to easily set their sweet spot in the resource over-provisioning vs. SLA violation trade-off. We motivate our scaling solution with analytical modeling and evaluation of the current Kubernetes behavior. The multi-forecast scaling engine and the proposed management parameter are evaluated both in simulations and with measurements on our collected Web traces to show the improved quality of fitting provisioned resources to service demand. We find that with just a few, but fundamentally different, and competing forecast methods, our auto-scaler engine, implemented in Kubernetes, results in significantly fewer lost requests with just slightly more provisioned resources compared to the default baseline.**

***Keywords*—Cloud computing, machine learning, auto-scaling, Kubernetes, forecast, resource management.**

I. Introduction

Web services and applications are typically cloud deployed, and underlying virtual resources are continuously adapted to fit the demand, e.g., the time varying amount of client requests hitting the application ingress. Web applications typically follow the microservice architecture, where the monolithic software is broken down into smaller independently managed components, usually realized by software containers. Allocating resources dynamically to constituent containers and scaling them properly on demand can be a challenging task. The scaling logic can be driven by various service management goals, e.g., either minimizing resource usage while sustaining a given service quality target, or minimizing Service Level Agreement (SLA) violations no matter the price paid for the provisioned resources. The decision-making in cloud scaling is further aggravated by the peculiar scaling behavior of the managed application, i.e., the function that translates the amount of requests to be served respecting the given SLA constraints to the necessary amount of resources to provision.

Kubernetes, the container orchestrator that has become the most popular cloud manager in the past few years, offers automatic scaling features, called Horizontal and Vertical Pod Autoscaling (HPA [1], VPA [2]). The vertical scaling feature of the Kubernetes system has been in beta version for years, and its major disadvantage is that currently it can only scale by terminating and restarting the application Pods [3]. Although there are proof-of-concept extensions of Kubernetes' VPA that make vertical scaling possible without restarts [4], this feature is not in the Kubernetes core yet. As the VPA is still under active development, major modifications are conceivable in the near future, therefore we focus on horizontal scaling. Pod represents the smallest deployment unit in Kubernetes encompassing typically a small number of tightly coupled containers. While custom and even external metrics are supported, by default HPA takes the metric of CPU usage into account, and makes threshold-based decisions automatically for scaling Pods out and in. This method obviously takes quasi-instant information and dictates rigid rules, but for most applications what HPA offers is acceptable. For cases though, in which incoming demand is hectic, SLA constraints are strict, and the budget for cloud resources is tight, one must exploit the possibility of custom metrics, and implement their own auto-scaler logic. This is the case in edge cloud setups that are characterized by limited infrastructure capacity and a relatively small number of clients, the requests of which are not multiplexed into a stable demand, while typical edge applications promise strict SLAs, e.g., low latency.

We see the following specific problems to address. 1) Scaling decisions are made solely on the current scaling period's observations, however, looking at a larger history might yield a better idea what comes next. 2) HPA parameters are set only once, so the scaling behavior is not adapted to current variability of the demand, although different request arrival regimes are well-known [5], [6]. 3) Even when applying default HPA, it seems that defining the scaling behavior is cumbersome, due to the numerous parameters to set. We therefore design an auto-scaler engine for Kubernetes that adapts

Manuscript received May 1, 2020; revised September 18, 2020 and November 25, 2020; accepted January 13, 2021. Date of publication January 19, 2021; date of current version March 11, 2021. Project no. 135074 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary under the FK_20 funding scheme. The associate editor coordinating the review of this article and approving it for publication was A. Dhamdhere. *(Corresponding author: Laszlo Toka.)*

The authors are with the MTA-BME Network Softwarization Research Group, Budapest University of Technology and Economics, 1111 Budapest, Hungary (e-mail: toka@tmit.bme.hu).

Digital Object Identifier 10.1109/TNSM.2021.3052837

scaling decisions to the actual demand, not constrained by limited information, or by the rigidity of the scaling logic. Moreover, we propose a simple management parameter, called *excess*, for the cloud tenant, i.e., application provider, to set the trade-off between resource over-, and under-provisioning.

Our contributions are the following. First, we evaluate HPA analytically with a loss-less MMPP/M/c queuing model, and numerically with simulating a discrete-time scaling method: we show through simulations how the proposed *excess* parameter would drive the operation of HPA. Second, inspired by the vast body of research performed in time series analysis and scaling decision-making in various systems ranging from power grids to cloud services, we propose a machine learning (ML)-based scaling method in order to capitalize on such well-known phenomena as daily and weekly profiles, and changing variability of request intensity throughout the day. Request arrival dynamics can be predicted with high accuracy taking into account historical information, thus we propose to use a forecast-based approach for a proactive scaling policy. Third, for it is shown in [7] that machine learning (ML)-based forecast methods perform differently depending on the characteristics of the time series, we apply four methods, i.e., auto-regressive (AR), unsupervised (HTM) and supervised (LSTM) deep learning, and reinforcement learning (RL) in our system, and make them compete. We implement their forecasts as Kubernetes custom metrics, and we dynamically switch between them and the default HPA based on a short-term back-testing plugin we call HPA+. We evaluate the quality of the ML-based forecasts and the scaling performance of HPA+ for a Web server deployment in Kubernetes, fed with the network traces we collected in a university campus, mimicking an edge scenario. We find that, on the one hand, the excess parameter plainly formulates the flexible resource overbooking vs. SLA violation balance, on the other hand, HPA+ leads to less request loss with a comparable amount of provisioned resources. Such a predictive scaling engine can be useful in an edge cloud environment to manage applications. In this case the incoming demand is hectic, SLA constraints are strict and available cloud resources are limited. Therefore it is important to use edge resources as efficiently as possible taking into account the SLA constraints. Our measurements and simulations show that a scaling engine that scales according to the dynamics of traffic is able to meet these expectations, and provide a more robust application, and better resource usage than the default HPA. The specific Web application that requires such an edge deployment could be a collaborative multi-user AR (Augmented Reality) game with strict constraints on the ultra reliable and low latency edge cloud backend.

The article is organized as follows. In Section II we summarize related work on scaling. In Section III we describe the quantitative relation between service demand and cloud resources for cloud-deployed applications. We present and evaluate analytic models we propose for describing HPA behavior in Section IV. In Sections V and VI we sketch the design of our HPA+, and evaluate its performance in an AWS-hosted Kubernetes cluster, respectively. We conclude this work in Section VII.

II. Related Work on Auto-Scaling

Since elasticity and dynamism are key concepts to cloud computing, offering appropriate application scaling is one of the most important features for a cloud provider. References [8], [9] give comprehensive surveys about scaling solutions applied in data centers. They categorize auto-scalers according to their underlying theoretical models. We follow suit, and highlight those recent works that fall close to our proposed solution.

Threshold-based: Authors of [10], [11] improved vertical elasticity in cloud systems of lightweight virtualization technologies with threshold-based scaling rules. Authors of [10] proposed ElasticDocker which supports vertical elasticity of Docker containers based on the IBM's autonomic computing MAPE-K principles. In [11] the authors presented the Resource Utilization Based Autoscaling System, which improved Kubernetes' VPA with the ability of dynamically adjusting the allocation of containers non-disruptively in a Kubernetes cluster. Both papers incorporated container migration and examined the possibilities in vertical scaling, while our proposed solution improves the horizontal auto-scaler. Khazaei *et al.* [12] presented the architecture of Elascale that provided auto-scalability and monitoring-as-a-service for any type of cloud software system, making scaling decisions based on an adjustable linear combination of CPU, memory, and network utilization. We argue that this simplification is a limiting factor in the scaling performance.

Queuing model-based: Several queuing models describe scaling systems by assuming a memoryless arrival process and an adaptive number of servers. Kaboudan [13] presented a discrete-time queuing model with a dynamic number of servers using a threshold-based scaling policy. They ran simulations, and compared their model to the behavior of an M/M/c queue. Mazalov and Gurtov [14] proposed a queuing model with an on-demand number of servers, which depended on the length of the queue, and they assumed a Poisson arrival process with a specified rate. Jia *et al.* [15] introduced a queuing model which used a threshold-based policy to better describe an industrial production process. The model assumed a Markov-modulated Poisson process (MMPP) as arrival, and the number of servers was selected from two predefined values based on the queue length. In all these cited papers we see the Markovian assumption of exponential inter-arrival times to be a limiting factor. Nevertheless, we also use an MMPP/M/c model in our analytic approximation of HPA.

Control theory-based: Using control theory, the behaviour of a dynamic system is changed by examining the output and reference values: the goal of the controller is to align the actual output to the reference based on the feedback to the input system. In an auto-scaling context, the reference value is the targeted SLA, and the output values to evaluate performance come from the system, e.g., CPU load or response time. For maximizing application QoS, while meeting both a time constraint and a resource budget, Zhu and Agrawal [16] developed a framework with Proportional-Integral (PI) control and reinforcement learning. They showed that their solution can efficiently scale the Virtual Machines (VMs) under the application with low overhead. Ali-Eldin *et al.* proposed two solutions for horizontal cloud elasticity: in their first work [17], the infrastructure was modeled as a G/G/N queue with variable N, which was used to design a reactive-adaptive proportional controller that acted based on the load dynamics, and could respond to sudden changes in it. They also managed to avoid oscillations and premature resource reduction; in their second article [18], two adaptive hybrid controllers were introduced that used both reactive and proactive control to change the number of VMs allocated to a service. Their system was analysed using different deployment scenarios with several real life workloads, and the proactive and reactive controller based scaling could outperform both the regression and the fully reactive scaling. Kalyvianaki *et al.* [19] used a Kalman filter to adjust the CPU allocation based on past utilization observations. The proposed system is able to scale multi-tier applications and has the ability to configure itself to any workload conditions. Baresi *et al.* [20] proposed a discrete-time feedback controller to scale resources both at VM-, and at container-level, and has the advantage of handling microservices-based applications. Their experiments showed that the proposed controller can outperform Amazon's Autoscaling significantly. Control theory-based models show high efficiency in application scaling. Several models combine other types of scaling methods, e.g., reinforcement learning, performance or demand prediction inside the controller to adapt to the application needs. Another advantage of these solutions is, that the controller can operate in the order of seconds as presented in [19], thus the scaling algorithm can respond quickly to changes in resource usage.

Reinforcement learning-based: Arabnejad *et al.* [21] combined two RL-based approaches: Q-learning and state-action-reward-state-action (SARSA) algorithms with a self-adaptive fuzzy logic controller that drove dynamic resource allocation for VMs. Horovitz and Arian [22] presented a threshold-based solution for horizontal container auto-scaling that used Q-learning to tune the scaling thresholds. Rossi *et al.* [23] proposed RL solutions (i.e., Q-learning, Dyna-Q, and model-based) in Docker Swarm that exploited different degrees of knowledge about system dynamics. In our auto-scaler engine we also leverage the power of RL methods.

Performance prediction-based: Wajahat *et al.* [24] proposed a neural network (NN)-, and regression-based application agnostic autoscaling solution, called MLscale. They used a multi-layer feed-forward NN to build the performance model of the application based on the request rate and system resource statistics. The model was able to predict the response time of the application. To predict the new response time after a proposed scaling action, they trained multiple linear regression models, which were able to determine the metrics from the current state taking into account the scaling decision. Rahman and Lama [25] proposed a solution to predict end-to-end tail latency of microservice-based applications and to scale based on that. They trained several ML methods, like random forest, linear regression, support vector regression and deep NN, to predict the latency of the application. For a given workload condition, they used the best model to find the highest resource utilization values of the relevant microservices, at which the given SLA targets would not be violated, and the scaling threshold was set to the found resource utilization value. Authors of [26] presented Microscaler, an autoscaling system that automatically detected SLA violations, determined the services requiring scaling, and evaluated how much resource those needed. They introduced Service Power, a metric that could be calculated from the response times of the service, and they used this metric to find the services which needed scaling. To determine the required instances for these services, they used Bayesian optimization and a step-by-step heuristic approach. Authors of [27] provided a solution called Autopilot, an ML-based horizontal and vertical scaler used by Google to configure the resources for tasks in a job. The primary goal was to reduce the difference between the limit and the actual resource usage (slack), while minimizing the risk that a task is killed. They used several ML techniques to predict the vertical resource limits based on historical data, and different ruled based approaches to horizontal scaling. In practice, Autopiloted jobs showed a slack of 23%, compared with 46% for manually-managed jobs.

Demand forecast-based: Short-term demand forecasting has been in the focus of researchers of various application domains for many years. In the energy sector, it is very important to know the power generated by wind turbines in advance. To solve this problem Li *et al.* [28] developed a 4-input NN which turned out to be better than the single parameter traditional approach. Catalão *et al.* [29] proposed a 3-layered feed-forward NN approach to forecast next-week electricity market prices. Their approach proved to be better than previously presented auto-regressive integrated moving average model (ARIMA) methods for the reason that it is less time consuming and easier to implement. For cloud computing, Chen [30] implemented a dynamic server provisioning technique to minimize the energy consumption of data centers. A custom AR method was presented in their study to forecast the number of connections on Windows Live Messenger servers. In [31] the authors applied signal processing and statistical learning algorithms to achieve online predictions of dynamic application resource requirements. Their forecast solution was based either on signatures of historic resource usage, or on a discrete-time Markov chain with a finite number of states, depending on whether the input traces showed repeating patterns or not. An effective prediction model was also provided by Islam *et al.* [32] for adaptive resource provisioning in the cloud. They evaluated several ML models, such as NNs and linear regression, on a dataset. They claimed that their solution was suitable for forecasting resource demand ahead of the VM instance setup time. The authors of [7] proposed an adaptive prediction method using genetic algorithms to combine time-series forecasting models, such as ARIMA, simple and extended exponential smoothing.

Many of the listed approaches modeled the resource needs of the application using ML techniques. In order to avoid both excessive over-provisioning and SLA violations, we use such performance models, and we also follow the ideas behind the line of work of demand forecast-based methods. In an earlier version of this work [33] we presented our solution that predicted incoming traffic and avoided SLA violation by scaling based on the application resource profile. In this article we extend that work with a study of HPA models with Markovian assumption regarding the incoming request rate. To this end we present an MMPP/M/c queuing model that allows to calculate the Pod usage assuming basic HPA operation; the model is validated by extensive simulations. Furthermore, besides the previously used real traces, we generate novel synthetic cloud application request traces by making use of an MMPP model that we fit to our real traces. The MMPP trace model then enables the creation and hence the evaluation of arbitrary volume of traffic, and the comparative analysis of our MMPP/M/c lossless model with our discrete-time lossy model of HPA. The created synthetic traces are also used to simulate the behaviour of our forecast methods in case of a less predictable traffic input. In addition to the forecast methods (AR, HTM, LSTM), presented in our prior paper, a Reinforcement Learning based method is also applied and incorporated in the simulation environment and in the proof-of-concept prototype. Our system strives to select the best performing one to control the scaling apparatus.

In Table I we present a comparison of the scaling methods listed in this section. Several important characteristics of the solutions were selected to be compared, the explanation of those can be seen at the bottom of the table. The quantitative comparison of these scaling engines is difficult due to the lack of standard testing frameworks. In most of the respective publications authors use synthetic or outdated traffic for validation, which does not fit with today's dynamically changing application usage, especially in an edge cloud environment. Another distinctive attribute, proactive operation has utmost importance, because in an environment where user demand is changing dynamically, those method have a better chance to avoid SLA violation and application overload. There are several methods operating with a proactive approach, but they differ in that the scaling takes place at the VM-, or the container level, and whether they support the scaling of distributed applications or not.

III. Cloud Application Resource Profiles

We dissect the problem of cloud auto-scaling into three phases. First, given the history of client requests' time series and multiple ML-based forecast methods applicable on them, we design a Kubernetes plug-in that implements the methods and selects the most reliable forecast at all times, based on the evaluation of their accuracy on recent measurements. Second, assuming a potential, but not completely certain request rate that the selected forecast method yields, we apply the *excess* parameter and adapt the application scale accordingly, in order to meet the application provider's intention on minimizing SLA violations. Third, we experimentally determine the required scale of the specific application given any request rate. This last step is in fact performed prior to the deployment by emulating arbitrary request rates. Here we delve into the third step, and we present the others in later sections.

Measuring the compute resource requirements of applications, i.e., resource profiling, has been investigated by researchers with resource optimization in mind. The authors of [34] for instance built application profiles in order to develop an application-execution policy that minimized the energy consumption of the mobile device. According to their approach, the applications in focus could be executed on the mobile device or in the cloud, and they built profiles for mobile applications based on the size of their input data, and the compilation deadline. In contrast to that work, our goal is to model the resource requirement of applications when serving any given rate of application requests, as accurately as possible. To this end we define application profiles as follows, projected to a Pod, i.e., the smallest runtime and scaling unit in the Kubernetes system. In a Pod we can include a container or a set of containers we want to run.

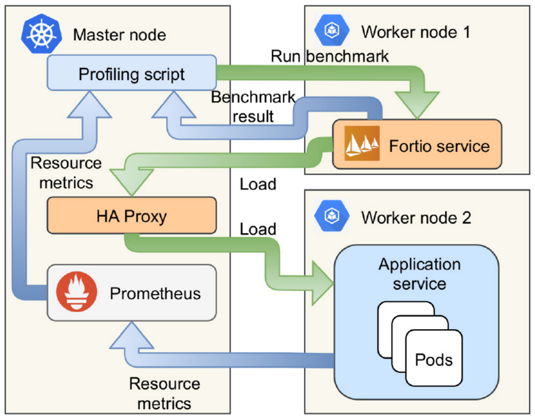


Fig. 1. Measurement setup for application profiling.

Definition 1 (Application Profile): The application profile is an $\mathrm{AP} : \mathbb{N}^+ \rightarrow \mathbb{R}^+$ function, which assigns the service rate per Pod to the number of active Pods.

We designed a measurement method to determine the profile for applications hosted in Kubernetes; the components of the measurement system are shown in Fig. 1. In our Kubernetes cluster we replaced the standard load balancer functionality of the Kubernetes Service object with an ingress controller. We deployed a Fortio [35] benchmark tool to a worker node, and the application we wanted to test was deployed to another worker node. During the measurement we run benchmark tests with an increasing number of requests per second. We started the benchmark with one Pod. When a test finished with the actual number of Pods, we increased the Pod count and started the measurement again. Each test took one minute. After each test, we collected the percentage of lost requests from fortio and the average CPU usage from Prometheus, a monitoring tool for Kubernetes [36].

We present results for 3 applications in Fig. 2 by showing the served query per second (QPS) in function of the number of running Pods. Using the empirical measurement results we can approximate the resource profile of each application. On the top diagram the empirical result of a Node.js application and an Nginx Web server can be seen. In the bottom figure the profile of an image classification application is shown. Let us take the measurement results of the Node.js Web application that serves static HTML files, depicted in Fig. 2 with the green

TABLE I
COMPARISON OF STATE-OF-THE-ART SCALING METHODS

Scaling system	Type	Adaptation	Proactive	Scaling	Unit	Validation	Distributed
Al-Dhuraibi et al. [9]	Threshold	X	X	V	C	+	X
Gourav et al. (RUBAS) [11]	Threshold	✓	X	V	C	+	X
Khazaei et al. (Elascale) [12]	Threshold	X	X	H	C	+	✓
Q. Zhu et al. [16]	Control theory	✓	X	V	VM	+	✓
A. Ali-Eldin et al. [18]	Control theory	✓	✓	H	VM	+	X
E. Kalyvianaki et al. [19]	Control theory	✓	X	V	VM	+	✓
L. Baresi et al. [20]	Control theory	✓	✓	H	VM+C	++	✓
Arabnejad et al. [21]	RL	✓	X	H	VM	++	X
Fabiana et al. [23]	RL	✓	X	H+V	C	unknown	X
Wajahat et al. (MLscale) [24]	Performance pred.	✓	✓	H	VM	++	X
Rahman et al. [25]	Performance pred.	✓	✓	H	C	+	✓
Yu et al. (Microscaler) [26]	Performance pred.	✓	X	H	C	+	✓
Rzadca et al. (Autopilot) [27]	Performance pred.	✓	X	H+V	C	unknown	✓
Gong et al. (PRESS) [31]	Performance pred.	✓	X	V	VM	++	X
Islam et al. [32]	Demand forecast	✓	✓	H	VM	+	X
Messias et al. [7]	Demand forecast	✓	✓	H	VM	++	✓
HPA+ (our system)	Demand forecast	✓	✓	H	C	++	X

Adaptation: historical data is used to better understand the application's resource demand
Proactive: scaling decisions are made based on predicted future demand or resource usage
Scaling: horizontal (H), vertical (V)
Unit: the unit of scaling is virtual machine (VM), or container (C), or both
Validation: ++ (real trace); + (generated trace with benchmark tools, or manually)
Distributed: whether the scaling system was designed to handle distributed applications, e.g., microservices

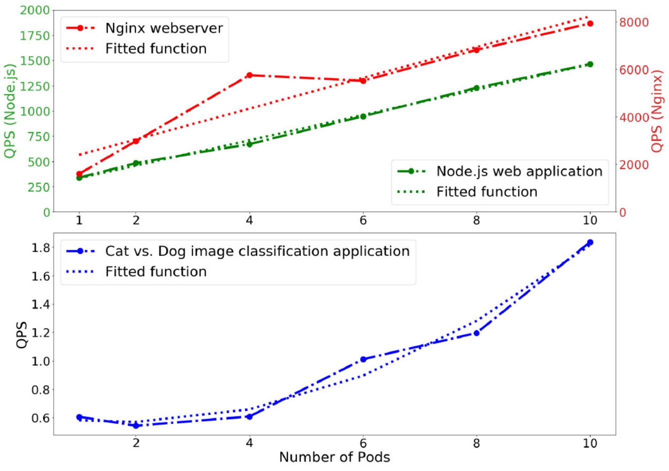


Fig. 2. Profiles of different applications.

dashed line. The empirical results can be approximated with the linear function (depicted with the green dotted line) of $125x + 209$, where x is the number of running Pods. We calculated the R^2 value (see in Section V-A) of this function and we got 0.997. R^2 is a statistical measure that represents the proportion of variance in the outcome variable that is explained by the predictor variables in the sample. Its value falls between 0 and 1; a value close to 1 means that our predictor fully explains the variance in our samples, i.e., the closer we are to 1, the better our prediction [37].

According to the fitted approximation function, the more active Pods are in the system, the lower service rate per Pod the application can provide. From this linear function the profile of the Node.js Web application using Def. 1 is $AP_{Node.js}(x) = 125 + \frac{209}{x}$. Measuring profiles for a wider set of applications is out of the scope of this article, but we note that there surely exist applications for which profiles are more complex than a linear relation, e.g., the measurements of the image classification application in the bottom part of Fig. 2 can be approximated by a quadratic function. We argue though that Kubernetes is predominantly used for Web applications, therefore selecting a popular Web server like Node.js for a representative profile seems evident. In the following we will use this function as the application profile in our simulations and in our Kubernetes deployments.

IV. Analytical Models of HPA

In this section we propose analytical models to describe the behavior of Kubernetes HPA. The models allow us to run extensive simulations for evaluating our ML-based forecast methods and their scaling accuracy against the baseline HPA.

A. HPA Operation in Details

In Kubernetes, HPA is influenced by several parameters. There are cluster level settings, e.g., down-scaling stabilization, Pod synchronization period and scaling tolerance, and there are HPA level parameters, such as minimum and maximum Pod number, scaling threshold according to arbitrary metrics. By default HPA is based on CPU utilization measurements: HPA periodically fetches monitoring data from the system, and takes a decision on how many Pods the cluster should have. This period is called a scaling interval. The scaling operation and the decision-making procedure is formally described in Def. 2.

Definition 2 (Kubernetes HPA Parameters and Operation): The following parameters must be configured for HPA [1]:

1) c_{min} and c_{max} are the minimal and maximal allowed Pod counts,
2) $\hat{u} \in [0, 1]$ is the targeted average CPU utilization over all running Pods,
3) scaling interval is a $\mathbb{R}^+$-long time window, at the end of which HPA evaluates metrics for scaling decisions,

4) $s \in [0,1]$ is the scaling tolerance.
5) *d* is the downscale stabilization, i.e., the duration the HPA has to wait before another downscale operation can be performed after the current one has completed.

Let d_i be an indicator function of whether downscaling is enabled or not during the scaling interval $i \in \mathbb{N}$.

$$d_i = \begin{cases} 1 & \text{if stabilization is active (downscale is disabled);} \\ 0 & \text{else.} \end{cases} \tag{1}$$

$u_i \in [0,1]$ is the measured CPU usage in scaling interval $i \in \mathbb{N}$. A scaling decision is made in scaling interval *i*, if

$$\left| \frac{u_i}{\hat{u}} - 1 \right| > s, \tag{2}$$

and the number of Pods in interval $i + 1$ is recursively yield:

$$c^*_{i+1} = \left\lceil c_i \frac{u_i}{\hat{u}} \right\rceil. \tag{3}$$

After applying the limits, Pods are terminated or instantiated to meet the count

$$c'_{i+1} = \begin{cases} c^*_{i+1} & \text{if } c_{min} \le c^*_{i+1} \le c_{max} \\ c_{min} & \text{if } c^*_{i+1} < c_{min} \\ c_{max} & \text{if } c^*_{i+1} > c_{max}. \end{cases} \tag{4}$$

HPA will not perform downscale operation if there was another one in the previous time window of length *d*.

$$c_{i+1} = \begin{cases} c_i & \text{if } c'_{i+1} < c_i \text{ and } d_{i+1} = 1; \\ c'_{i+1} & \text{else.} \end{cases} \tag{5}$$

The higher the *d* value, the more resources the application uses, because it scales out without such stabilization, but scaling in is slower compared to applying a low *d*.

B. An MMPP/M/c Model for Changing Arrival Regimes

For modeling purposes we make some simplification to HPA's operation and to the application profile. We assume that *d* is equal to the scaling interval, i.e., $d_i = 0 \ \forall i$, and $c_{min} = 1$, $c_{max} = \infty$, $s = 0$. The application profile is assumed to be constant, i.e., $AP(x) = \mu$, μ being the rate of the exponentially distributed service times. We further suppose that the arrival process of requests can be described as a Markov-modulated Poisson process (MMPP) with *m* states; we denote the respective transition matrix by *Q* and the *m* arrival rates by $\lambda_1, \lambda_2 \ldots, \lambda_m$.

HPA sets the number of servers periodically, so in our model we make the case for a changing number of servers that is adapted dynamically to the change of the arrival rate. HPA is reactive, i.e., the change of arrival rate takes effect on the number of servers in the next scaling interval. Therefore the number of servers follows the MMPP arrivals, but lags one scaling period. Let $\lambda(t)$ be the arrival rate at time *t*. By assuming that the time MMPP spends in a particular state (e.g., minutes) is orders of magnitude longer than the time between request arrivals (e.g., milliseconds), the number of servers at *t* can be calculated with the following formula:

$$C(t) = \left\lceil \frac{\lambda(t-1)}{\rho\mu} \right\rceil, \tag{6}$$

where ρ is the server utilization used in M/M/c queue models, and can be calculated as $\rho = \lambda/(c\mu)$ with λ arrival rate, *c* servers and μ service rate. The utilization equals to the management parameter of HPA, i.e., $\rho = \hat{u}$.

From this model we can calculate Pod usage to describe the behaviour of Kubernetes HPA, and we can analyze the MMPP parameters. The steady-state probabilities π of the underlying Markov chain of MMPP arrivals are given by:

$$\pi = \pi Q \text{ and } \pi\mathbb{I} = 1, \tag{7}$$

where $\mathbb{I}$ is an all-ones vector. The steady-state arrival rate is:

$$\lambda_{MMPP} = \pi\lambda^T \tag{8}$$

where λ^T is the transpose of the row vector $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_m)$.

Although using the steady-state distribution of MMPP we can infer the arrival rates, and from that the probability of having a given number of Pods in the system, the disadvantage of this model is that it does not account for job losses. As timed-out requests are paramount to count during the operation, we also propose a more complex model of HPA.

C. Our Discrete-Time Model for HPA

We propose a discrete-time queuing model for mimicking the calculation of c^*_{i+1} in HPA in subsequent simulations. Our model is similar to the one presented in [13], where the scaling decision was based on the ratio of customers waiting and on the number of servers. Although instead of the number of queued requests, we use the number of served requests in the scaling decision, because the number of served requests and the CPU usage are tightly coupled, and the latter is the basis of scaling decisions of HPA.

Let the number of requests in the system in period *i* be

$$L_i = L_{i-1} - M_{i-1} + \Lambda_i - T_i \tag{9}$$

where Λ_i is the number of arrived requests, M_i is the number of served requests in period *i* and T_i is the number of queued requests that expire in period *i*. M_i is defined as follows.

$$M_i = \min\{c_i AP(c_i), L_i\} \tag{10}$$

where c_i is the number of Pods and $AP(c_i)$ is the service rate described by the Application profile defined in Definition 1, i.e., M_i is the minimum of the number of requests in the system and of the number of requests the system could serve in period *i*. L_i contains the requests arrived during time period *i*, but does not contain the requests served in period $i-1$, nor the requests lost in period *i*. We assume an empty queue in the initial period, i.e., $i = 0$, $L_0 = 0$, $c_0 = 1$, $\Lambda_0 = 0$.

This model expresses request loss with the number of timed-out requests T_i, reflecting under-provisioning of the system in the particular scaling interval. The formula for T_i is given in (11), where *t* denotes the value of timeout expressed in scaling intervals, e.g., if $t = x$, then all requests arrived in period $(i - x)$ and not served from period $(i - x)$ until period $(i - 1)$ are timed out in period *i*.

$$T_i = \begin{cases} 0, & \text{if } (i-t) < 0 \\ \left(L_{i-t} - \sum_{j=1}^{t} M_{i-j}\right)^+ \end{cases} \tag{11}$$

CPU usage can be approximated by the ratio of number of served requests over the maximum number of requests that can be served. Therefore the current CPU usage can be written in the following form:

$$u_i \simeq \frac{M_i}{AP(c_i)c_i}, \tag{12}$$

which yields the following for (3), the number of required Pods in Kubernetes:

$$c^*_{i+1} = \left\lceil \frac{M_i}{\hat{u}AP(c_i)} \right\rceil. \tag{13}$$

We introduce downscale stabilization as a positive integer d: if $d = 1$, it has no impact on the system, because scaling operation can not be performed within one scaling interval. The indicator function of downscale stabilization in scaling interval i can be formed as follows for $d > 1$:

$$d_i = \begin{cases} 1 & \text{if } \exists j : 2 \leq j \leq d, c_{i-j+1} < c_{i-j} \\ 0 & \text{else.} \end{cases} \tag{14}$$

Later, during our measurements, the downscale stabilization is set to the length of the scaling interval in order to make its effects negligible. With this model we can simulate the number of Pods in the system, their resource usage and the number of lost requests, taking into account the unique profile of the application and our *excess* parameter.

D. Evaluation of the MMPP/M/c and Discrete Models

There are widely used data sets for testing Web applications, the two most common are NASA-HTTP [5] (two months' worth of all HTTP requests to the NASA Kennedy Space Center Web server in Florida) and WorldCup98 [6] (three months' HTTP requests made to the 1998 World Cup Web site). We find that these logs are rather outdated (1995 and 1998). Therefore to validate the presented HPA models, we ran simulations with recent anonymized Web traffic traces collected in a university campus network for a week in the middle of a fall semester. The dataset contains the flow count per second to Facebook; the trace is shown in Fig. 3. Since the Facebook data is generated by real users, it is a good example of how the traffic of a cloud application manifests, e.g., it shows daily trends, and the average number of requests per minute is not extremely high, but it shows relatively high variance, hence it is a good fit for the edge cloud application flavor, i.e., relatively small user base, hectic traffic. Therefore we use these traces for our simulations and experiments.

We also generated an MMPP-based traffic trace for simulation purposes. An MMPP has been fitted to the Facebook traffic trace using the algorithm described in [38]. The algorithm is able to calculate the parameters of an MMPP from the traffic information. For simulations we used a generated arrival sequence from this fitted MMPP; the final MMPP-based traffic can be seen in Fig. 4. With this synthetic trace, we are able to compare the MMPP/M/c HPA model with the lossy discrete model. Moreover, it is apparent that the synthetic trace does not show the daily pattern as clearly as the real trace, therefore our forecast methods' (in Section V) behavior can be simulated and evaluated in case of a less predictable traffic, with which we can analyze the robustness of our system.

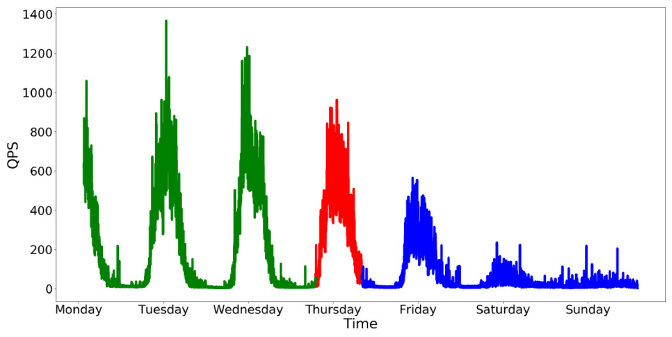


Fig. 3. One week Facebook traffic in a university campus (green: training data, red: test data, blue: weekend, not used).

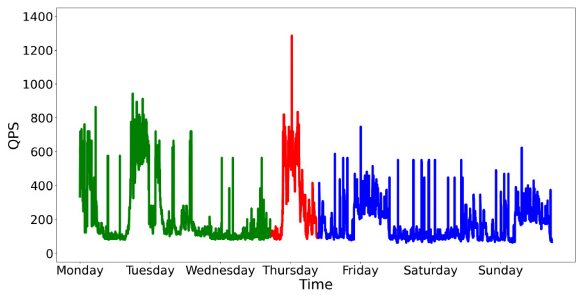


Fig. 4. One week of MMPP-based traffic generated from the Facebook load using the algorithm described in [38] (green: training data, red: test data, blue: weekend, not used).

First we checked whether the discrete model matches HPA's behaviour by comparing the number of Pods started in response to the real trace in a simulator of the model, and in a real Kubernetes cluster. With our model we could simulate the number of Pods in the system, i.e., resource usage, and the number of expired requests, taking into account the unique profile of the application. The scaling interval was set to one minute making the start-up time of new Pods negligible in comparison. The downscale stabilization was set to one minute, too. Results are shown in Fig. 5: x-axis shows the time, the y-axis of the first diagram depicts the number of Pods, that of the second diagram shows the sum of Pod minutes, i.e., cumulative sum of Pod numbers accumulated through the simulated minutes. As the results suggest, the discrete model behaved similarly to the real HPA operation, i.e., it gave a good approximation of the number of Pods. We calculated the mean squared error (MSE) on the per minute Pod number of the simulated model compared to the measurement, and we got 0.27 while the average number of Pods is 2.38 by the discrete model and 2.28 by the HPA measurement.

Next we compared the discrete and the MMPP/M/c-based models. Since the latter requires MMPP input, we used our synthetic traffic (Fig. 4). We applied the following equation to calculate the number of Pods for the MMPP/M/c-based model in scaling interval $i : c_{i+1} = C(i+1)$, where C is the method of calculating Pods given in (6). We run the simulations with a constant service rate and a downscale stabilization of less than one scaling interval. The results are shown in Fig. 6. Like in Fig. 5, the top chart depicts the number of Pods, the bottom chart shows the cumulative sum of Pod minutes in function

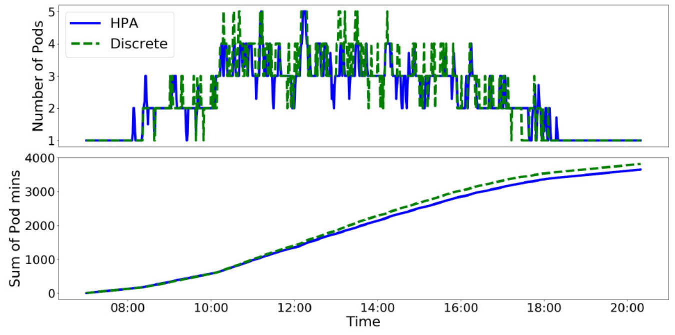


Fig. 5. Comparison of the lossy discrete model and HPA - Pod usage.

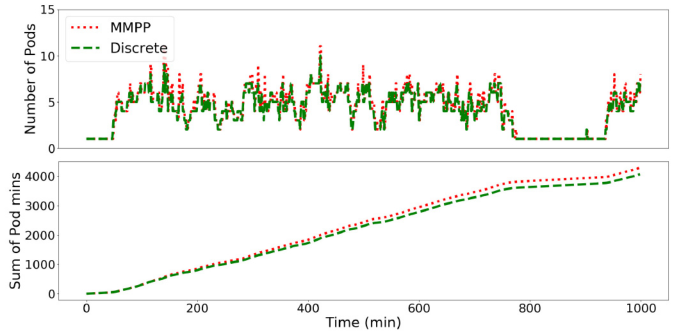


Fig. 6. Comparison of lossy discrete and lossless MMPP/M/c models - Pod usage.

TABLE II
COMPARISON OF LOSSY DISCRETE AND LOSSLESS MMPP MODELS

Pod usage	Excess parameter		
	0.85	0.9	0.95
Average - lossy discrete	7.18	5.19	4.07
Average - lossless MMPP/M/c	7.31	5.33	4.30
MSE	0.16	0.19	0.29

of time. The two analytical models behave similarly. Slight difference can be observed when the input traffic is volatile. To quantify the difference, we calculated the average number of Pods and the MSE from the results, and we listed them in Table II. Based on the results for various *excess* parameter values (that we define later in Section VI), it can be stated that the MSE is low compared to the average number of Pods.

It follows that, under certain conditions, the MMPP/M/c model can be used instead of the discrete model, enabling the analytical evaluation of the behavior of HPA. Since the MMPP/M/c model assumes lossless operation, intuitively the approximation of HPA is better in settings that result in fewer lost requests, which is reached by lower values of the *excess* parameter, hence the relatively low MSE value at 0.85.

V. ML FOR SCALING: THE PROACTIVE HPA+

In order to achieve more effective resource usage and higher service quality than the reactive HPA, we decided to create a scaling method that anticipates future requests and allocates or releases resources in advance. In our so-called HPA+ solution incoming request prediction and the resulting scaling decisions are implemented in two separate modules. In contrast to many solutions that propose using only Q-learning for server scaling [39], [40], [41], we use popular ML methods for prediction [30], [32] in addition to the Q-learning method: auto-regression and NNs. We find it important to utilize models that have substantial differences in their operation. AR is a simple model with low resource requirement, whereas LSTM and HTM require more computing power. However, the latter two models differ in nature: the former requiring supervised learning and the latter being an unsupervised learning approach. Furthermore, the RL method (Q-learning) requires a relatively long time for exploring the state space until good actions (or predictions) are issued.

A. *ML-Based Prediction of Web Requests*

Our ML models were trained and tested on the anonymized Web traffic traces shown in Fig. 3. We chose to use the time series of requests targeting Facebook, because the traces show typical usage patterns: it is a highly visited website, the number of visits from the campus shows a daily and weekly profile, and the standard deviation of visits per minute is high, which describes well the request dynamics of an average Web service with a moderate customer base. To build and evaluate forecast models we standardized the dataset using Z-score normalization. We chose the time granularity to be in the order of minutes because both the Web request time-outs and the Pod startup times fall in the order of seconds. Furthermore, using a grid search on the granularity value and other system parameters, we found that the trends in traffic can be best captured using one minute granularity. This observation is also supported by Rattihalli *et al.* [11] in their work.

Our first model to predict the traffic load was an auto-regressive (AR) model. It assumed that the traffic load at a given time linearly depended on the previous values of the time series. After the training phase we realized that the previous 32-minute observations had to be used for achieving the best accuracy. The coefficients to what extent we use each previous observation was calculated by optimizing the model.

The second model for the time series analysis was chosen from supervised deep learning methods: we used the popular Long Short-Term Memory (LSTM) NN. We examined several values for the size of the look-back window and the most beneficial turned out to be 15-minute long. Thus for LSTM we used a 15-minute look-back window, i.e., all the load values from $t - 15$ to t were used to predict the load in $t + 1$. We also used the load difference between loads in the look-back window and the number of elapsed minutes from midnight in each t as input to the network.

We picked our third model from the family of unsupervised deep learning: we selected the widely used Hierarchical Temporal Memory (HTM). It is a biological NN, designed to learn temporal patterns from sequences [42]. We used the implementation of Numenta.org in which we encoded the input load and the timestamp so that the network would take into account 15-minute history. Numerous parameters of the HTM architecture were optimized to achieve the best performance.

We used SARSA and Q-learning as our fourth, RL-based approach. In this case, the state of the system is modeled with the current load and the number of running Pods. The RL approach looks for the best action to execute in the given state. The possible actions in a state are: scale out, scale in, idle. Even though the output of the RL solution is the number of Pods required, we can convert this to a predicted load by calculating the QPS served by the given number of Pods.

The training set for the four models consisted of the first 3 days of the week and the test set was the fourth day, as shown in Fig. 3. Each model was trained on the training set. The hyperparameters of the models were optimized using grid search and walk-forward cross validation, which means that each model was re-trained several times on increasingly larger sets. Then with the best performing hyperparameters the final models were created and then evaluated on the test set. For an illustrative example, we wanted our forecast models to learn the daily profile of workdays; weekends (and Friday afternoon) are significantly different, but the four-day window proved to be sufficiently long for training and testing.

The results of the forecast methods were examined with the root MSE (RMSE) of their prediction and with their R^2 value. The R^2 value assesses how well a model explains and predicts future outcomes. The closer the value is to 1, the stronger the predictive power. The RMSE of the AR, LSTM, HTM and RL were 0.092, 0.107, 0.138 and 0.737 and the R^2 value were 0.879, 0.859, 0.819 and 0.456. Comparing the time needed for training, the AR model was significantly faster than the others, its training time was $14.7ms \pm 0.6ms$. Besides the advantage of being taught quickly, computing the AR model is not resource intensive, but it is sensitive to outliers. On the other hand, LSTM handles outliers well, but its training is more resource intensive ($283s \pm 12s$). In terms of resource requirements, the HTM falls between AR and LSTM ($13s \pm 2s$), however the main advantage of HTM is that it is robust to noise and it is able to learn continuously from each new input pattern, no prior training is required. The important features of RL-based approaches are learning without prior knowledge, and the ability to learn online and update environmental knowledge by actual observations. However the learning time of our RL model is similar to that of LSTM, i.e., $212s \pm 24s$.

B. The More ML, the Better

We further compared the instantaneous performance of our lossy discrete-time queuing model of HPA and that of our ML forecasting models in numerical simulations. We compared them to the oracle: perfectly predicting the number of arriving requests in every scaling interval, and setting the number of Pods accordingly. The oracle is used as a benchmark for resource usage, i.e., we evaluated each model through the excess of their Pod usage, compared to that of the oracle. Furthermore, on the other side of the provisioning decision problem, we looked at the request loss ratio, denoted by Loss in the figures, which shows the ratio of lost requests to the total number of Web requests. Again, our four ML-based models (AR, LSTM, HTM, RL) were trained on the 3-day series of flow counts towards facebook.com in our traffic traces, and then the models were evaluated on the fourth day's traffic.

The results of the simulations for all five scaling methods can be seen in Fig. 7. On the top diagram the cumulative request loss ratio (Loss) is displayed. At the beginning of the simulation, HTM performed best in terms of Loss. In the middle of the day the cumulative loss ratio of the models changed dynamically, and at the end of the day AR finished with the lowest value. RL gave the worst Loss performance throughout the whole day. On the bottom plot the cumulative excess of Pod usage (relative to the oracle) is shown. HTM is far from the optimal resource usage: the y-axis shows that it uses around 8% more Pod minutes than the oracle. The same applies to the RL method: except for a few cases throughout the day, it cannot give better prediction than the rest of the models. The other three methods perform similarly to each other: at the beginning AR is the closest to the optimal, but in the second half of the simulation LSTM gets closer to the optimal. The results show that although LSTM is closer to the oracle considering the Excess of Pod usage, it leads to higher loss than what is obtained by the other methods. However, those other methods all use slightly more Pods.

We also run the comparative simulations with the synthetic MMPP trace. Results are shown in Fig. 8: the AR and the LSTM models lead head-to-head in terms of Loss. The hectic MMPP traffic could not be well tracked by the HTM model and therefore it led to overprovisioning. The scaling decisions of RL were delayed or insufficient, thus it resulted in many lost requests again. The lossy discrete HPA model allocated far less resource than necessary, leading to a relatively high amount of losses, so it performed much worse in terms of the number of lost requests than the first two models.

The results demonstrate that there is no single method that would always approximate well the optimal scaling. In fact, as the input traces show varying dynamics throughout the day, methods perform well or poorly episodically. Therefore we design our framework so that models are extendable and system parameters can be changed at any time. An arbitrary number of models can be applied; although more models require more resources, with different models the system can cover different cases of input traffic fluctuations. So we

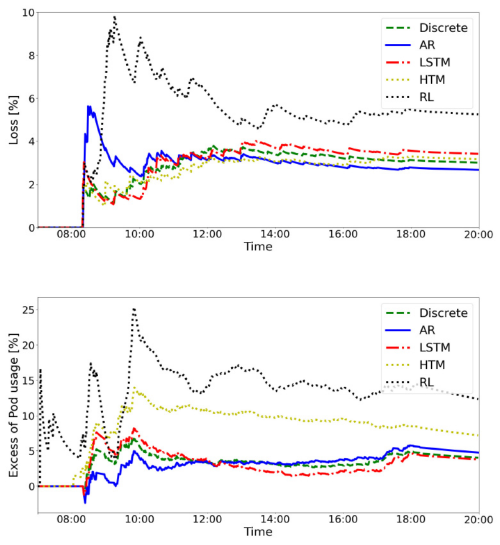
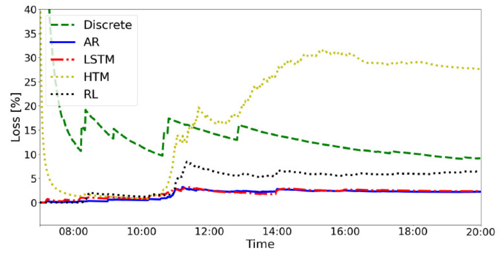


Fig. 7. Simulations of discrete-time HPA and of the ML-based forecast models.



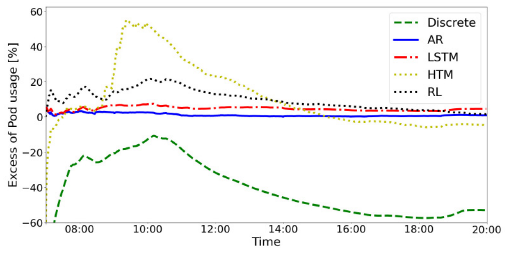


Fig. 8. Simulations of HPA's discrete-time and of the ML-based forecast models on the generated MMPP traffic.

decided to create a scaling engine, in which several methods compete with each other and the active, decision-making method is chosen based on its performance on recent input.

C. The Proactive Scaling Engine: HPA+

We therefore designed the high level operation of our scaling engine, called HPA+, as follows:

1) it contains four ML-based forecast models that predict that rate of incoming Web requests;
2) each minute, it calculates the accuracy of the forecast models for the recent past, and selects the best one;
3) based on the prediction of the selected model, it calculates the number of required Pods (using the profile of the application) for the next minute;
4) if the accuracy of all the ML-based models are poor, it switches back to the default HPA operation temporarily.

Each model's accuracy is calculated based on their past predictions in a brief history window, and the prediction of the best performing model is used for the scaling decision. The evaluation is performed on the average of the Relative Percentage Differences (RPD) in the last n minutes, i.e.,

$$\frac{1}{n}\sum_{i=t-n}^{t-1}\left(2\frac{|\hat{q}_i - q_i|}{\hat{q}_i + q_i}\right), \tag{15}$$

where q_i stands for the average QPS in the ith minute and $\hat{q}_i$ is the predicted average QPS for the ith minute. The closer a model's accuracy is to 0, the better its prediction, so from the four ML models, the one with the lowest RPD value will be used each time. We used a parameter for the fallback to HPA, called fallback threshold. If the accuracy value of our models given by (15) is greater than this threshold, then the engine switches back to the default HPA operation.

Before running experiments, we simulated the operation of HPA+ to examine whether better scaling decisions could be obtained than with HPA, the default auto-scaler in Kubernetes.

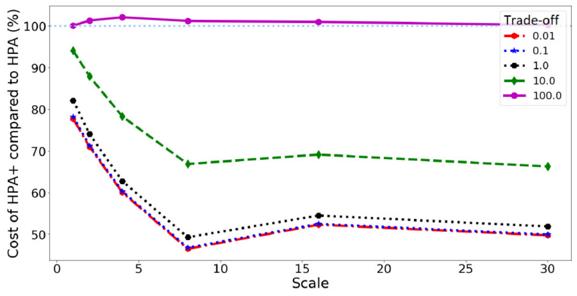


Fig. 9. Proportion of HPA+ and HPA costs as a function of input load scale.

In order to simulate the racing behaviour of HPA+, for different parameter sets, i.e., history window and fallback threshold, we trained the ML models and the HPA discrete model on the 3-day dataset and evaluated them on the fourth day. The active model was determined every minute by (15). The number of pods used and the number of lost requests per minute was calculated considering the scaling decisions of the selected active model only. To account for non-deterministic behavior, we performed this simulation 25 times for each parameter set.

After the parameter search using grid search, we found that a history window of 5 minutes, and 0.3 as fallback threshold should be used in HPA+ for our experiments that we present in the next section. In addition to the optimal values of the history window and of the fallback threshold, we also examined how the efficiency of HPA+ would change if the input load was scaled up: we ran simulations in which input request rate was scaled up to 30x of the original trace. We compared the cost induced by HPA+ and by HPA; the cost contains the price of Pod minutes and the penalty for lost requests. We summarize these two cost terms with a weighting factor, called as *trade-off* parameter, which translates the cost of SLA violations, i.e., the number of lost requests, into the cost of cloud resources, i.e., Pod minutes. In Fig. 9 we show the average ratio of total cost of HPA+ over that of HPA on the y-axis, while the x-axis depicts the scale of the input request rate intensity compared to the original trace. We show the cost ratio for 5 different *trade-off* parameters: 0.01, 0.1, 1, 10, 100. The results suggest that HPA+ incurs slightly higher costs than HPA only if cloud resources are significantly more expensive than SLA violations, i.e., *trade-off* parameter is 100. In other cases HPA+ reaches significant savings, mostly owing to heavily reduced request losses with HPA+. Moreover, HPA+ leads to a significantly cheaper operation when the input request rate is high, i.e., scales over 5x. This phenomenon is due to the granularity of Pods adapted to the request rate: Pods become relatively smaller in capacity compared to the overall demand, hence the number of Pods can be better fitted to the load input.

VI. Evaluation of HPA+ and the Excess Parameter

In this section we give details about the implementation of HPA+ in Kubernetes, we introduce an easy-to-use management parameter that the application provider can tune depending on their *trade-off* parameter, and finally we present the promising measurement results of our experiments with HPA+.

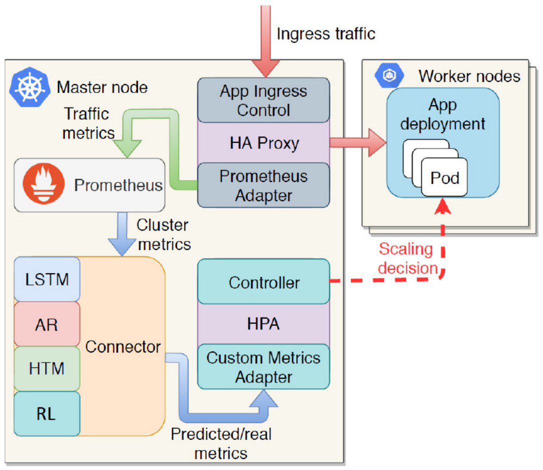


Fig. 10. HPA+ system architecture.

Algorithm 1 Main HPA+ Control Loop

```
Require: all ML models are trained
Require: Profile(x): an invertible Application Profile function
    (see Definition 1)
 1: $excess \Leftarrow$ value of excess parameter from Section VI-B
 2: $MLModels \Leftarrow$ list of all ML models
 3: while TRUE do
 4:     compute accuracy for all models in $MLModels$
 5:     if at least one model is accurate, see (15) then
 6:         for all $model$ in accurate MLModels do
 7:             get prediction from model
 8:         end for
 9:         $pred \Leftarrow$ select best prediction according to (15)
10:         $pod \Leftarrow$ get # of Pods (Prometheus)
11:         $newPods \Leftarrow Profile^{-1}(\frac{pred}{pod})/excess$
12:         send $newPods$ to Prometheus as custom metric
13:     else
14:         switch back to default HPA
15:     end if
16:     sleep 1 minute
17: end while
```

A. HPA+ Implementation

For the experiments we created a Kubernetes deployment in Amazon Web Services (AWS). We used Kubernetes Operations (kops) [43], an open-source tool for deploying and maintaining production grade, highly available Kubernetes clusters on AWS. Our cluster contained one master and five worker nodes. All of them were EC2 m5.xlarge instances, i.e., each instance had 4 vCPUs and 16 GB RAM.

We implemented HPA+, our ML-based system, making use of application resource profiles, described in Section III, and the AR, LSTM, HTM, and RL models, described in Section V-A. The scaling engine was implemented in Python and was run as a daemon service on the Kubernetes' master node.

The architecture and workflow of the system can be seen in Fig. 10. As the test application, we used a Node.js Web application that could be accessed through an HAProxy ingress controller [44] that served as the load balancer for the service. HPA+ heavily relies on metric collection for which we use the Prometheus monitoring system [36]. When the Node.js application is under load, the HAProxy Prometheus Adapter captures the relevant metrics, e.g., request count, response count, errors, and Prometheus collects them. The main part of the HPA+ system is the Connector module. It collects QPS metrics from Prometheus and, with the use of the AR, LSTM, HTM, and RL models, it predicts future QPS values. Based on the Node.js application profile, it calculates the number of Pods that could serve the requests thus proactively scales out or in the Pods in the system.

If the parameters of the forecast ML models and of the HPA+ system had not been optimized beforehand, at the beginning of the operation the ML models start training on the QPS data. During this phase, the system can operate properly utilizing online learning models, such as HTM, and default system-wide parameters. Although those latter, i.e., fallback threshold, history window, time granularity of decisions, should be optimized for the specific application input rate dynamics. Once sufficient amount of historical data is collected, one can create a simulation of the past traffic and thereafter can optimize the system-wide parameters. HPA+ can always make a fallback to the default HPA mechanism that uses CPU usage as scaling metric. When the models are ready to forecast, the system switches to the ML-based scaling, if their accuracy proves to be sufficiently good. In operation, the last minute's average QPS is appended to the models' history and a prediction is made for the next minute's QPS based on historical values. The forecast models' accuracy is continuously evaluated in HPA+, so they work in a racing environment: before predicting the QPS for the next minute, each model's accuracy is calculated using (15), and the best performing one is used, as explained in Section V-C. The predicted metrics and, in case of a fallback, CPU metrics are reported by the Connector to HPA as custom metrics. The vanilla HPA module then makes the scaling decision based on those.

The pseudo-code of the main control loop of HPA+ is described in details in Algorithm 1.

B. Adaptive Operation Under Excess Policies

An application provider that uses Kubernetes or a similar management system to run its application pays for the reserved cloud resources, while it provides QoS to its users according to the SLA. The application provider may have various strategies depending on the cost of running the application and the penalties due to SLA violations. They can try to comply with the SLA as much as possible no matter how much it will cost, which results in significant over-provisioning of resources. Alternatively they can strive to operate the service as cheaply as possible, at the expense of frequent SLA violations. In the midway, it is possible to seek an operational point with reasonably provisioned resources, violating SLA rarely.

We introduce a generic management parameter, called *excess* parameter, to describe the resource allocation strategy of the application provider. Let the *excess* parameter be a number between 0 and 1. We chose 3 illustrative values of this parameter that we used in the experiments:

1) Conservative strategy (excess = 0.85): scaling actions are triggered to keep resource utilization at 85% on average. This strategy is wasteful in terms of resources, but leads to a negligible amount of SLA violation;
2) Normal strategy (excess = 0.9): a compromise that keeps a balance between utilization and SLA violation. Scaling is performed to keep resource utilization around 90%.
3) Best effort strategy (excess = 0.95): a resource-saving strategy that scales only if the system is under heavy load (keeping the utilization at 95% on average), so SLA violations are certain, but running the application is close to the cheapest possible in terms of allocated resources.

We experimentally compared the operation of HPA and HPA+ and evaluated the *excess* parameter's effect on them. The *excess* parameter is matched with the resource utilization rate, e.g., the application provider wants the scaling to be performed so that the utilization rate of all the Pods in the system is at the value of this parameter on average.

C. Results of HPA+ Experiments

We tested the implemented HPA+ in a real life scenario with the different scaling strategies and compared the results to those of the native HPA. We generated input traffic corresponding to our Facebook traces shown in Fig. 3: the green parts were the training data for the models (Monday, Tuesday and Wednesday) and the red was the testing data (Thursday). We collected the number of application Pods, arrived requests and lost requests every minute. This measurement was repeated for each scaling policy (Conservative, Normal, Best effort) 5 times, and we calculated the average results from them.

Since one pays the cloud provider for the reserved resources, proportional to the number of running Pods, and SLA violation compensation fees for lost requests to customers, we examined these exact two metrics. In Fig. 11 the cumulative percentage of lost requests (left y-axis) and sum of Pod minutes (right y-axis) can be seen in function of time for both scaling engines, under 3 *excess* policies. The percentage of lost requests is plotted in log scale. The Sum of Pod minutes value shows how much it would cost to run the service until a given moment on the x axis, if the 1-minute usage of a Pod cost $1.

As shown in Fig. 11, there is no significant difference in Pod usage between HPA and HPA+, however it is clear that the lower *excess* parameter value we have, the fewer lost requests the system will see. Even at its highest level, with the Best effort scaling policy, HPA+ leads to a drastically lower number of lost requests than HPA.

The final comparative results of HPA and HPA+ operations can be seen in Table III. In the first column the three scaling policies are listed. The second column shows the reduction in loss with HPA+ compared to HPA, the third column shows the Pod minutes of HPA+ compared to HPA. For example in case of Normal strategy, during the HPA+ operation the number of lost requests is reduced by 44%, but the overall Pod minutes were only 3% higher than in case of HPA. HPA+ used 9% more Pods in case of the Best effort strategy, but the number of lost requests were reduced by 72%. The drop in the number of lost requests is due, among others, to the competitive operation of forecast methods in HPA+. From the five scaling methods (CPU-based HPA, AR, LSTM, HTM, and RL predictions), the scaling engine strives to use the one which suits the actual traffic the best. In Fig. 12 the five scaling methods are shown with the points in time when they were active under the Normal scaling policy. At the beginning and at the end of the experiment there were fallbacks to HPA, but during the day the forecast methods alternated to adapt to the dynamically changing traffic. In this particular measurement AR ran 44%, LSTM 32%, HTM 16% and default HPA 8% of the time, not leaving airtime for RL.

We found that the number of lost requests can be reduced at the expense of a slightly increased Pod usage with HPA+. Comparing different scaling strategies, we observed that the proposed *excess* parameter correctly controls the resource utilization in the experiments for both HPA+ and HPA, so

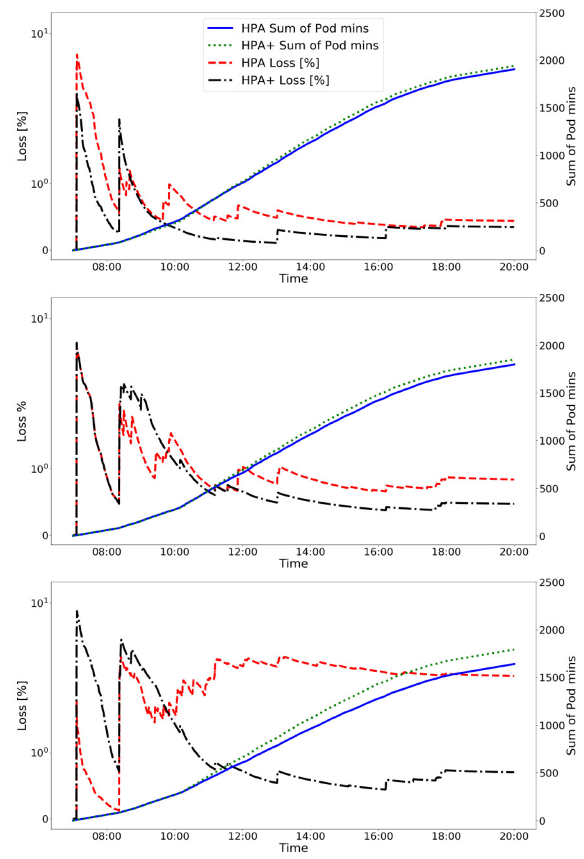


Fig. 11. Comparison of HPA and HPA+ with Conservative (top), Normal (center), Best effort (bottom) scaling strategies.

TABLE III
LOST REQUESTS AND RESOURCE USAGE OF HPA+ RELATIVE TO THOSE OF HPA

Policy	Lost requests	Resource usage
Conservative	-22%	+2%
Normal	-44%	+3%
Best effort	-72%	+9%

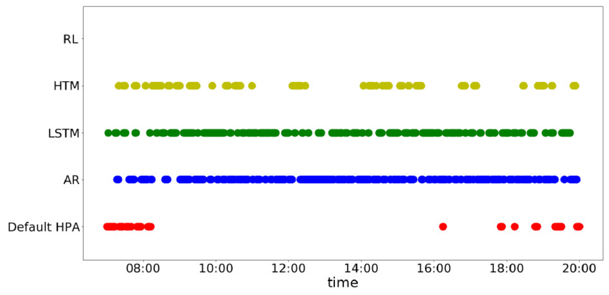


Fig. 12. Activated forecast/scaling methods during the day.

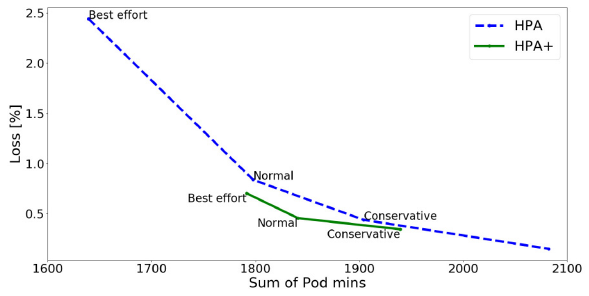


Fig. 13. Sum of Pod mins and loss for various scaling policies.

this parameter is suitable for describing the desired resource utilization of the application. Although, the same scaling policy may result in different resource usage in case of HPA and HPA+. To fully analyze our scaling engine compared to HPA, we examined the case when the resource usage of HPA and HPA+ were similar (even when applying two different *excess* parameters) and compared the loss. In Fig. 13 the Sum of Pod mins (resource usage) and the request loss can be seen for all three scaling policies for both HPA and HPA+. It can be concluded that even in case of similar resource usage, the number of rejected requests of HPA+ is smaller than HPA's. It means that the competitive operation of HPA+ results in more efficient management of the same amount of resources.

Besides the default HPA, it is important to evaluate the performance of HPA+ with that of other state-of-the-art scaling engines described in Section II. However, the evaluation of their performance is difficult in most cases due to the lack of implementation code, or detailed explanation about the exact algorithm, or the parameter settings therein. Moreover, several of those methods scale only VMs, or scale containers but only vertically, which renders quantitative comparison challenging. Therefore, we selected two methods that scale containers horizontally and we implemented their algorithms in Python, then we run their simulations with our Facebook traces. In order to cover fundamentally different solutions, we chose a threshold-based (Elascale [12]) and a reinforcement learning-based method [21]). The results of the simulations are shown in Fig. 14: HPA+ provided better performance in terms of number of lost requests with similar resource usage. In case of Elascale, the reason can be that during scaling, only one instance is started or terminated, therefore it can not handle the hectic user demand fast enough. The reinforcement learning-based algorithm is close to the performance of HPA+, but the advantage of our solution is that it selects from different models during the operation, therefore it can slightly outperform the single RL algorithm.

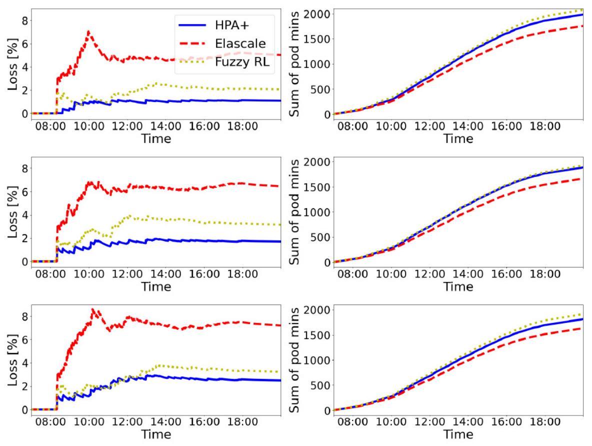


Fig. 14. Comparison of HPA+, Elascale [12] and one of the Fuzzy RL algorithms [21] with different scaling strategies: Conservative (top), Normal (center), Best effort (bottom).

VII. Conclusion

The concept of microservices and corresponding technologies provide a new way of application development and service operation. Crucial management tasks, such as resource scaling, are delegated to the underlying cloud platforms, but application providers provisioning the novel services over cloud platforms need appropriate configuration interfaces to police the operation. We addressed Kubernetes, the most widely used container orchestration system, and evaluated several auto-scaling methods. First, the currently available horizontal auto-scaler was investigated: two analytic models were proposed to capture the characteristics of the broadly used method. The models can play important role in the planning phase when the underlying resource pool is designed for a given customer base and SLA level. Second, we designed and implemented a novel proactive scaling engine including multiple ML-based forecast methods in order to optimize the operation in different circumstances. The proposed solutions were evaluated via simulations and real measurements where the input traffic was constructed from traces collected from a campus network. Third, we focused on easy configurability, and introduced a simple management configuration, called *excess* parameter, to control the resource allocation policy by specifying the targeted compromise between resource over-provisioning and SLA violation. We found that our auto-scaling engine is able to significantly decrease the number of rejected requests, at the cost of slightly higher resource usage. However, the additional resource consumption is negligible compared to the regular operation. Furthermore, by setting the *excess* parameter so that HPA and HPA+ consumed similar amount of resources, our auto-scaling engine provided a lower lost request ratio.

References

[1] *Horizontal Pod Autoscaler*, Kubernetes, Mountain View, CA, USA. 2020. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[2] *Vertical Pod Autoscaler—Kubernetes*. Accessed : Jan. 29, 2021. [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

[3] *Vertical Pod Autoscaling*, Google Kubernetes Engine, Mountain View, CA, USA, 2021. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler

[4] *Huawei-Cloudnative/Kubernetes at Vertical-Scaling*. Accessed : Jan. 29, 2021. [Online]. Available: https://github.com/huawei-cloudnative/kubernetes/tree/vertical-scaling

[5] *NASA-HTTP Logs*. Accessed : Jan. 29, 2021. [Online]. Available: ftp://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[6] *WorldCup98 HTTP Logs*. Accessed : Jan. 29, 2021. [Online]. Available: ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html

[7] V. R. Messias, J. C. Estrella, R. Ehlers, M. J. Santana, R. C. Santana, and S. Reiff-Marganiec, "Combining time series prediction models using genetic algorithm to autoscaling Web applications hosted in the cloud infrastructure," *Springer Neural Comput. Appl.*, vol. 27, no. 8, pp. 2383–2406, 2016.

[8] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

[9] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 430–447, Mar./Apr. 2018.

[10] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, 2017, pp. 427–479.

[11] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, 2019, pp. 33–40.

[12] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," in *Proc. 27th Annu. Int. Conf. Comput. Sci. Softw. Eng. (CASCON)*, 2017, pp. 234–240.

[13] M. A. Kaboudan, "A dynamic-server queuing simulation," *Comput. Oper. Res.*, vol. 25, no. 6, pp. 431–439, 1998.

[14] V. V. Mazalov and A. Gurtov, "Queuing system with on-demand number of servers," *Mathematica Applicanda*, vol. 40, no. 2, pp. 1–12, 2012.

[15] Y.-H. Jia, L.-X. Tang, Z. G. Zhang, and X.-F. Chen, "MMPP/M/C queue with congestion-based staffing policy and applications in operations of steel industry," *Springer J. Iron Steel Res. Int.*, vol. 26, no. 7, pp. 659–668, 2018.

[16] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," *IEEE Trans. Services Comput.*, vol. 5, no. 4, pp. 497–511, 4th Quart., 2012.

[17] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth, "Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control," in *Proc. ACM 3rd Workshop Sci. Cloud Comput. (ScienceCloud)*, 2012, pp. 31–40.

[18] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proc. IEEE Netw. Oper. Manag. Symp. (NOMS)*, 2012, pp. 204–212.

[19] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters," in *Proc. ACM 6th Int. Conf. Auton. Comput. (ICAC)*, 2009, pp. 117–126.

[20] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 217–228.

[21] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proc. 17th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. (CCGRID)*, 2017, pp. 64–73.

[22] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *Proc. 6th IEEE Int. Conf. Future Internet Things Cloud (FiCLOUD)*, 2018, pp. 82–92.

[23] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. 12th IEEE Int. Conf. Cloud Comput. (CLOUD)*, 2019, pp. 329–338.

[24] M. Wajahat, A. Gandhi, A. A. Karve, and A. Kochut, "Using machine learning for black-box autoscaling," in *Proc. IEEE 7th Int. Green Sustain. Comput. Conf. (IGSC)*, 2016, pp. 1–8.

[25] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, 2019, pp. 200–210.

[26] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Serv. (ICWS)*, 2019, pp. 68–75.

[27] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at Google," in *Proc. 15th EuroSys Conf.*, 2020, pp. 1–16.

[28] S. Li, D. C. Wunsch, E. A. O'Hair, and M. G. Giesselmann, "Using neural networks to estimate wind turbine power generation," *IEEE Trans. Energy Convers.*, vol. 16, no. 3, pp. 276–282, Sep. 2001.

[29] J. P. S. Catalão, S. J. P. S. Mariano, V. M. F. Mendes, and L. A. F. M. Ferreira, "Short-term electricity prices forecasting in a competitive market: A neural network approach," *Elect. Power Syst. Res.*, vol. 77, no. 10, pp. 1297–1304, 2007.

[30] G. Chen, "Energy-aware server provisioning and load dispatching for connection-intensive Internet services," in *Proc. 5th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2008, pp. 337–350.

[31] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive elastic resource scaling for cloud systems," in *Proc. IEEE 6th Int. Conf. Netw. Serv. Manag. (CNSM)*, 2010, pp. 9–16.

[32] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Elsevier Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 155–162, 2012.

[33] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive AI-based autoscaling for Kubernetes," in *Proc. IEEE/ACM Int. Symp. Clust. Cloud Internet Comput. (CCGrid)*, 2020, pp. 599–608.

[34] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *Proc. IEEE INFOCOM*, 2012, pp. 2716–2720.

[35] *Fortio*. Accessed : Jan. 29, 2021. [Online]. Available: http://fortio.org/

[36] *Prometheus*. [Online]. Available: https://prometheus.io/

[37] J. Miles, "R squared, adjusted R squared," in *Wiley StatsRef: Statistics Reference Online*. Hoboken, NJ, USA: Wiley, 2014.

[38] A. Rajabi and J. W. Wong, "MMPP characterization of Web application traffic," in *Proc. 20th IEEE Int. Symp. Model. Anal. Simulat. Comput. Telecommun. Syst. (MASCOTS)*, 2012, pp. 107–114.

[39] J. Rao, X. Bu, C.-Z. Xu, L. Y. Wang, and G. G. Yin, "Vconf: A reinforcement learning approach to virtual machines auto-configuration," in *Proc. ACM 6th Int. Conf. Auton. Comput. (ICAC)*, 2009, pp. 137–146.

[40] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow," in *Proc. Int. Conf. Auton. Auton. Syst. (ICAS)*, 2011, pp. 67–74.

[41] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Wiley Concurrency Comput.*, vol. 25, no. 12, pp. 1656–1674, 2013.

[42] J. Hawkins and S. Blakeslee, *On Intelligence: How A New Understanding of the Brain Will Lead to the Creation of Truly Intelligent Machines*, New York, NY, USA: Macmillan, 2007.

[43] *kOps—Kubernetes Operations*. Accessed : Jan. 29, 2021. [Online]. Available: https://github.com/kubernetes/kops

[44] *HAProxy*. Accessed : Jan. 29, 2021. [Online]. Available: https://www.haproxy.com/

László Toka (Member, IEEE) received the Ph.D. degree from Telecom ParisTech in 2011. He worked with Ericsson Research from 2011 to 2014. He is an Assistant Professor with the Budapest University of Technology and Economics, a Vice-Head of HSNLab, and a member of both the MTA-BME Network Softwarization and the MTA-BME Information Systems Research Groups. His research focuses on cloud computing and artificial intelligence.

Gergely Dobreff received the Graduation degree (with Highest Hons.) in computer science in 2019. He is currently pursuing the master's degree with the Budapest University of Technology and Economics. He works on sports analytics and autoscaling research projects. He specializes in machine learning and data science.

Balázs Fodor received the B.Sc. degree in computer science from the Budapest University of Technology and Economics in 2019, where he is currently pursuing the master's degree. He works on research projects related to autoscaling, cloud, microservices, and machine learning.

Balázs Sonkoly received the M.Sc. and Ph.D. degrees in computer science from BME in 2002 and 2010, respectively, where he is an Associate Professor and the Head of MTA-BME Network Softwarization Research Group. He has participated in several EU projects and he was the demo Co-Chair of ACM SIGCOMM in 2018, EWSDN in 2014 and 2015, IEEE HPSR 2015. His current research activity focuses on cloud/edge computing, NFV, SDN, and 5G.