Adaptive AI-based auto-scaling for Kubernetes

Laszlo Toka, Gergely Dobreff, Balazs Fodor, Balazs Sonkoly
MTA-BME Network Softwarization Research Group
Budapest University of Technology and Economics
Hungary
Email: {toka,dobreff,fodor,sonkoly}@tmit.bme.hu

***Abstract*—Kubernetes, the prevalent container orchestrator for cloud-deployed web applications, offers an automatic scaling feature for the application provider in order to meet the ever-changing amount of demand from its clients. This auto-scaling service, however, requires a seemingly difficult parameter set to be customized by the application provider, and 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. Therefore we set the ultimate goal of making cloud-based web 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 AI-based 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, as soon as possible. We also introduce a compact management parameter for the cloud-tenant application provider in order to easily set their sweet spot in the resource over-provisioning vs. SLA violation trade-off.

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 competing forecast methods, our auto-scaling engine, implemented in Kubernetes, results in significantly less lost requests with slightly more provisioned resources compared to the default baseline.

***Keywords*—cloud computing, artificial intelligence, 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. This latter represents the changing load induced by 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 on the side of provisioned resources. The cloud scaling decision-making 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 and VPA) [3]. Pod represents the smallest deployment unit in Kubernetes encompassing typically one or a small number of tightly coupled containers. While custom and even external metrics are supported, by default the metric of CPU usage is taken into account, and threshold-based decisions are made automatically for scaling up and down. This method obviously takes only quasi-instant information and dictates rigid rules; but for most of the applications the offer of default HPA 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 offer strict SLAs.

We see the following specific problems to address. 1) Scaling decisions are made solely on the current scaling period's observations, by default on average CPU usage, however, looking at a larger history might yield a better idea what comes next. 2) HPA parameters are set only once initially, so the scaling behavior is not adapted to current variability of the demand, although different request arrival regimes are well-known [5], [7]. 3) Even if applying default HPA, due to the numerous parameters to be set, it seems that defining the scaling behavior is cumbersome. We therefore design an auto-scaler engine for Kubernetes that adapts scaling decisions to the actual demand, not constrained by information available to 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 as they wish.

Our contributions are the following. First, we model HPA with a discrete-time scaling method, and through simulations we validate it against the operation of HPA. Second, inspired by the vast body of research performed in time series analysis and scaling decision-making in various systems from power grids to cloud services, we propose a more complex scaling method than that of HPA. As queuing model and reinforcement learning-based policies assume Markovian request arrival, we turn to AI-based methods in order to capitalize on such well-known phenomena as daily and weekly profiles, and changing variability of request amount throughout the day. We propose to use a forecast-based approach for a proactive scaling policy. Third, as it is shown in [26] that AI-based forecast methods perform differently depending on the characteristics of the time series, we apply three methods, i.e., AR, HTM and LSTM, in our approach. We implement their forecasts as Kubernetes custom metrics, and we dynamically switch between them and the default HPA, based on a short-term backtesting plugin we call HPA+. HPA+ always selects the metric that best describes the recent user demand. We evaluate the quality of the AI-based forecasts and the scaling performance of HPA+ for a web server deployment in Kubernetes, fed with the network traces collected in a university campus. 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 resources.

The paper is organized as follows. In Sec. II we categorize related work on scaling, depending on the method based on which scaling decisions are made. In Sec. III we describe the quantitative relation between service demand and cloud resources for the most common web applications and for an image classifier application. We present and evaluate the analytic model we propose for describing the HPA behavior in Sec. IV. In Sec. 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 Sec 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. [24], [9] give comprehensive surveys about scaling solutions applied in data centers. They categorize auto-scalers according to their underlying theoretical models or methods. In this section we follow suit, and highlight those recent works that fall close to our proposed solution.

Threshold-based: Authors of [8], [16] improve vertical elasticity in cloud systems of lightweight virtualization technologies with threshold-based scaling rules. Authors of [8] propose ElasticDocker which supports vertical elasticity of Docker containers autonomously, based on the IBM's autonomic computing MAPE-K principles. In [16] the authors present the Resource Utilization Based Autoscaling System, which improves Kubernetes' VPA with the ability to dynamically adjust the allocation of containers non-disruptively in a Kubernetes cluster. Both papers incorporate container migration and examine the possibilities in vertical scaling, while our proposed solution improves the horizontal autoscaler. Khazaei et al. [22] present the architecture and initial implementation of Elascale that provides 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.

Reinforcement learning-based: Arabnejad et al. [10] combined two reinforcement learning (RL) approaches: Q-learning and State-action-reward-state-action (SARSA) algorithms with a self-adaptive fuzzy logic controller that drives dynamic resource allocations for virtual machines (VMs). Horovitz et al. [18] present a threshold-based solution for horizontal container auto-scaling that uses Q-learning to tune the scaling thresholds. Rossi et al. [15] propose RL solutions (i.e., Q-learning, Dyna-Q, and Model-based) in Docker Swarm that exploit different degrees of knowledge about system dynamics.

Queuing model-based: Several queuing models describe automatic scaling systems by assuming a memoryless arrival process and an adaptive number of servers. Kaboudan et al. [21] 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 a classic M/M/c queue. Mazalov et al. [25] proposed a queuing model with an on-demand number of servers, which depends on the length of the queue, and they assumed a Poisson arrival process with a specified rate. Jia et al. [20] 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 works we see the Markovian assumption of exponential inter-arrival times to be a limiting factor.

Forecast-based: Short-term demand forecasting has been in the focus of researchers for 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. [23] developed a 4-input neural network which turned out to be better than the single parameter traditional approach. Catalao et al. [12] proposed a 3-layered feedforward neural network approach to forecast next-week electricity market prices. Their approach proved to be better than previously presented autoregressive integrated moving average model (ARIMA) methods for the reason that it is less time consuming and easier to implement. For cloud computing, Chen et al. [13] implemented a dynamic server provisioning technique to minimize the energy consumption of data centers. A custom auto-regression (AR) method was presented in their study to forecast the number of connections on Windows Live Messenger servers. Using their solution a significant amount of energy can be saved. In [30] 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. [19] for adaptive resource provisioning in the cloud. They evaluated several machine learning models such as neural networks and linear regression on a dataset. They claimed that their solution is suitable for forecasting resource demand ahead of the VM instance setup time. The authors of [26] proposed an adaptive prediction method using genetic algorithms to combine time-series forecasting models, such as ARIMA, simple and extended exponential smoothing. We follow the idea behind this line of work, and apply forecast-based methods, out of which our proposed system strives to select the best fitting one for controlling the scaling apparatus.

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 AI-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 measure the required scale of the specific application given any request rate. This last step can be in fact performed prior to the deployment by emulating arbitrary request rates. In this section we delve into the details of 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 goal in mind. The authors of [29] 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.

Definition 1. *[Application Profile] The application profile is an* $AP{:}\mathbb{N}^+ \rightarrow \mathbb{R}^+$ *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 on top of 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 [1] benchmark tool to one of the worker nodes, 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

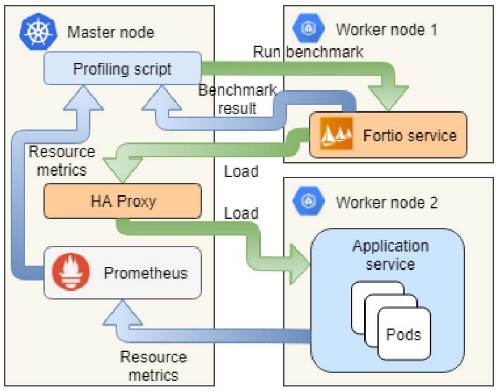


Fig. 1. Measurement setup for application profiling

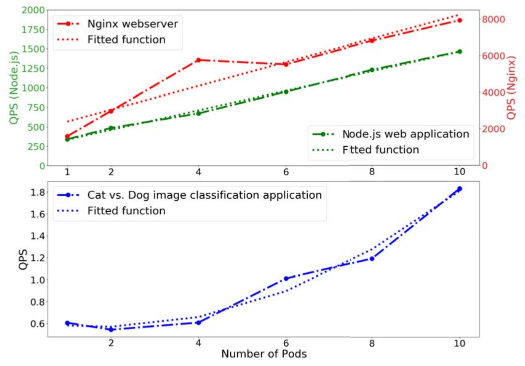


Fig. 2. Profile of different applications

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 [6].

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 simple Node.js application and an Nginx webserver can be seen. In the bottom figure the profile of an image classification application is shown. As an example, measurement results on the profile of a simple Node.js web application that serves static HTML files, are depicted in Fig. 2 with a green dashed line. The empirical results can be approximated with the linear function (depicted with a green dotted line) of $125x + 209$, where x is the number of running Pods. We calculated the R^2 value (see in Sec. V-A) of this function and we got 0.997. R^2 is a statistic that represents the proportion of variance in the outcome variable which is explained by the predictor variables in the sample. Its value is between 0 and 1, 1 means that our predictor fully explains the variance in our samples, i.e. the closer we are to 1, the better our prediction. [27].

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 paper, but we note that there surely exist applications for which the profile is more complex than a linear relation, e.g., the measurements of the image classification application 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 Kubernetes deployments.

IV. A DISCRETE-TIME QUEUING MODEL OF HPA

In the following we propose an analytical model to describe the behavior of Kubernetes. Our main purpose with an HPA model is that it allows us to run extensive simulations for evaluating our AI-based forecast methods and their scaling accuracy against the baseline HPA.

A. HPA operation in details

In Kubernetes, HPA scaling is influenced by several parameters. There are cluster level settings like down-scaling stabilization, Pod synchronization period and scaling tolerance, and there are HPA level parameters such as minimum and maximum Pod number, scaling threshold which can work according to different 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 until a next possible scaling action. 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 are configured by the operator of HPA [3]:*

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) $\mathbb{R}^+$ *time length of a scaling interval, when 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* $i \in \mathbb{N}$*.*

$$d_i = \begin{cases} 1 & \textit{if stabilization is active (downscale is not enabled)}; \\ 0 & \textit{else}. \end{cases} \tag{1}$$

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

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

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

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

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

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

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

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

The higher the d value, the more resources the application uses, because it scales up regardless, but scaling down is slower compared to applying a low d.

B. Our discrete-time model for HPA

We propose a discrete-time queuing model for mimicking the calculation of c^*_{i+1}. Our model is similar to the one presented in [21], where the scaling decision was based on the ratio of customers waiting and 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 base of scaling decisions of HPA.

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

$$L_i = L_{i-1} - M_{i-1} + \Lambda_i - T_i \tag{6}$$

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

$$M_i = min\{c_i AP(c_i), L_i\} \tag{7}$$

where c_i is the number of Pods and $AP(c_i)$ is the service rate described by the Application profile defined in Def. 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 reflects the number of requests arrived during time period i, and the number of served requests in period $i-1$ similarly to [21], minus the number of lost requests in period i. We assume 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 (8), where t denotes the value of timeout expressed in scaling intervals, e.g., if $t = x$, then all the requests arrived in period $(i-x)$ and not served from period $(i-x)$ until period $(i-1)$ are timed out in period i.

$$T_i = \begin{cases} 0, \text{ if } (i-t) < 0 \\ (L_{i-t} - \sum_{j=1}^{t} M_{i-j})^+ \end{cases} \tag{8}$$

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

$$u_i \simeq \frac{M_i}{AP(c_i)c_i}, \tag{9}$$

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

$$c^*_{i+1} = \left\lceil \frac{M_i}{\hat{u}AP(c_i)} \right\rceil. \tag{10}$$

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

$$d_i = \begin{cases} 1 & \text{if } \exists j : 2 \leq j \leq d, c_{i-j+1} < c_{i-j} \\ 0 & else. \end{cases} \tag{11}$$

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 examine and simulate the number of pods in the system, resource usage and number of lost requests, taking into account the unique profile of the application and scaling stabilization.

C. Evaluation of our discrete-time model

There are widely used datasets 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 [7] (three months' HTTP requests made to the 1998 World Cup Web site). We find that these logs are rather outdated (1995 and 1998) and might not reflect the current usage of microservice-based web services. Because of these reasons in order to validate the presented HPA models, we ran our simulations with 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 for Facebook traffic; the trace is shown in Fig. 3.

First we checked whether the discrete model matches HPA's behaviour, so we compared the number of Pods started in response to the input 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

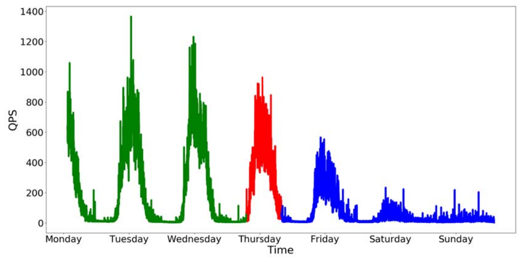


Fig. 3. The one week traffic load of Facebook in a university campus (green: training data, red: test data, blue: weekend, not used)

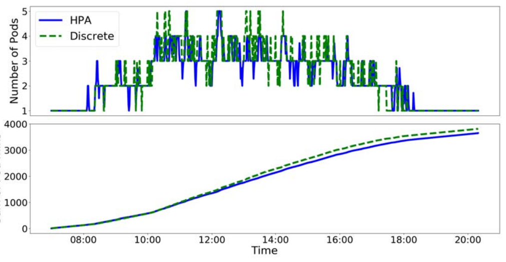


Fig. 4. Comparison of Discrete model and HPA - Pod usage

one minute making the start-up time of new Pods negligible in comparison. The downscale stabilization was set to one minute, too. Results of simulations are shown in Fig. 4: x-axis shows the time, the y-axis of the first diagram shows 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 HPA, i.e., it gave a good approximation to 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.

V. AI 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 Q-learning for server scaling [28], [14], [11], we use popular machine learning methods for prediction [13], [19]: auto-regression and neural networks. It was important to utilize models that have substantial differences in their operation. AR is a very simple model with very low resource requirement, whereas HTM and LSTM require much more computing power. However, the latter two models differ in nature, with one model providing supervised learning and the other modeling unsupervised learning.

A. AI-based prediction of web requests

Our AI 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. The time granularity of minute was chosen because both the web request time-outs and the Pod startup times fall in the order of seconds.

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) neural network. We examined several values for the size of the look-back window and the most beneficial turned out to be 15 minutes. Thus for LSTM we used a 15-minute look-back window, i.e., all the load values from $t-15$ to t was 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 methods: we selected the widely used Hierarchical Temporal Memory (HTM). It is a biological neural network which was designed to learn temporal patterns from sequences [17]. 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. The numerous parameters of the HTM architecture were optimized to achieve the best performance.

The training set for the three models consisted of the first 3 days of the week and the test set was the fourth, as shown in Fig. 3. For illustrative example, we wanted our forecast model 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 mean squared error (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 and HTM were 0.092, 0.107 and 0.138 and the R^2 value were 0.879, 0.859 and 0.819. Comparing the time needed for training, the AR model was significantly faster than the others, the training time is $14.7ms \pm 0.6ms$. 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 ($283sec \pm 12sec$). In terms of resource requirements, the HTM falls between AR and LSTM ($13sec \pm 2s$), 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, meaning that no prior training is required.

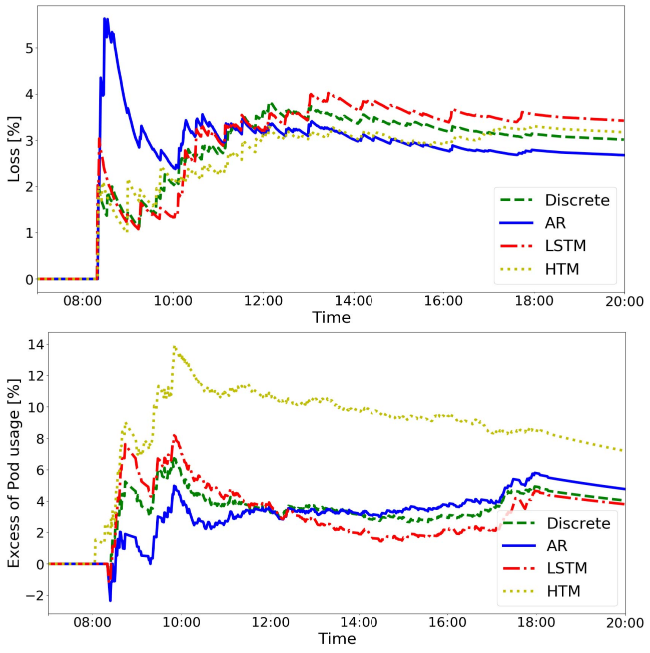


Fig. 5. Simulations of HPA's discrete-time model and of the AI-based forecast models

B. The more AI, the better

We evaluated the performance of our discrete-time queuing model of HPA and that of our AI-based forecasting models in numerical simulations. We compared them to the perfect forecast, i.e., in every scaling interval the number of arriving requests is assumed to be *a priori* known, and the number of Pods is precisely set accordingly. The perfect forecast is used as a benchmark for resource usage, i.e., we evaluated each model through the excess of Pod usage, compared to the perfect forecast's. Furthermore, on the other side of the provisioning decision problem, we looked into the request loss ratio, denoted by Loss, which shows the ratio of lost requests to the total number of web requests.

Again, our three AI-based models (AR, LSTM and HTM) 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 four scaling methods can be seen in Fig. 5. 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, and AR was the worst one. 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. On the bottom plot the cumulative relative Pod usage in excess compared to the perfect forecast is shown. HTM is far from the optimal scaling's resource usage and the y-axis shows that it uses around 8% more Pod minutes than the perfect forecast. The other three methods perform simirarly to each other: at the beginning AR is the closest one 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 perfect forecast considering the Excess of Pod usage value, it leads to higher loss than what is obtained by the other methods. However, those other methods all use slightly more Pods.

From the results we concluded that there is no single method that would always lead to optimal scaling. In fact, as the input traces show variable dynamics, selected methods perform better or worse compared to each other. That is why we decided to create a scaling engine, in which several such methods compete with each other and the active method is chosen based on its performance on recent input.

C. The proactive scaling engine: HPA+

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

1) it contains three AI-based forecast models that predict that rate of incoming web requests;
2) each minute, it calculates the accuracy of the forecast models, 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 the models is poor, it switches back to the default HPA operation.

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.,

$$\frac{1}{n}\sum_{i=t-n}^{t-1}\left(2\frac{|\hat{q}_i - q_i|}{\hat{q}_i + q_i}\right), \tag{12}$$

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 three AI models, the one with the lowest RPD value will be used each time. We used a threshold parameter for the fallback to HPA, called fallback threshold. If the accuracy value of our models given by (12) 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. In order to simulate the racing behaviour of HPA+, for each parameter set, history window and fallback threshold, we trained the AI models and the HPA discrete model separately on the 3-day dataset and evaluated them on the fourth day. The active model was determined in each minute based on (12).

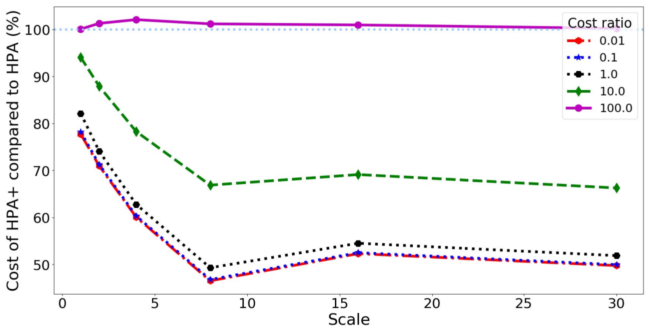


Fig. 6. Proportion of HPA+ and HPA costs as a function of input load scale

The number of pods used and the number of lost requests of each minute was calculated considering the scaling decisions of the active model only. Since LSTM and HTM are not deterministic, we performed this simulation 25 times for a given parameter set.

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 traces. We compared the cost induced by HPA+ and by HPA; the cost contains the price of Pod minutes and the penalty of lost requests. We summarize these two cost terms with a weighting factor, called as cost ratio 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. 6 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 compared to the original traces. We show the cost ratio for 5 different cost ratio 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 violation, i.e., cost ratio 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.

We optimized the n parameter, i.e., the history window, and the fallback threshold on the training set for the original traces, and we used $n = 5$ and 0.3 as fallback threshold in our experiments that we present in the next section.

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 cost ratio parameter, and finally we present the promising measurement results of our experiments of HPA+.

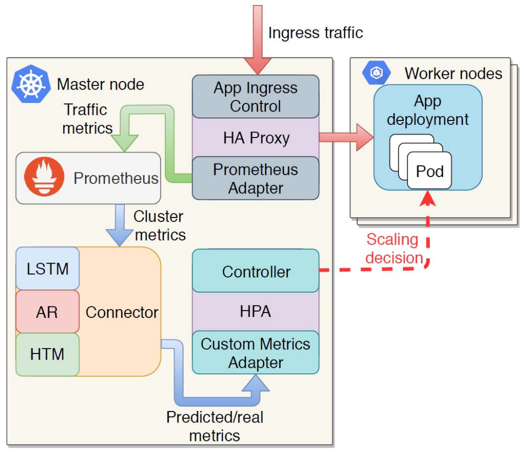


Fig. 7. HPA+ system architecture

A. HPA+ implementation

For the experiments we created a production grade Kubernetes deployment in Amazon Web Services (AWS). For the cluster creation we used Kubernetes Operations (kops) [4], an open-source tool for deploying and maintaining production grade, highly available Kubernetes clusters on AWS. For our measurements we used a cluster with 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 AI-based system, making use of application resource profiles, described in Sec. III, and the AR, LSTM and HTM models, described in Sec. V-A. The architecture and workflow of the system can be seen in Fig. 7. For the test application we used a Node.js web application that could be accessed through an HAProxy ingress controller [2] that served as the load balancer for the service. HPA+ heavily relies on metric collection so we used the Prometheus monitoring system [6] for that. When the Node.js application was under load, the HAProxy Prometheus Adapter captured the relevant metrics, e.g., request count, response count, errors, and Prometheus collected them.

The main part of the HPA+ system is the Connector module. Its responsibility is to collect QPS metrics from Prometheus and, with the use of the AR, LSTM and HTM models, to predict future QPS values. Based on the Node.js application profile, it calculated the number of Pods that could serve the requests thus proactively scaled up or down the number of Pods in the system.

Although the parameters of the models had been already optimized, at the beginning of the measurement the AI models started training on the QPS data. As 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). Obviously while the models were being trained, the HPA+ could not produce prediction based scaling metrics, so HPA+ made a fallback to the default HPA mechanism that uses CPU usage as scaling metric. When the models were ready for prediction the system switched to the AI-based scaling, if their accuracy proved to be sufficiently good.

The forecast models' accuracy is continously evaluated in HPA+: the last minute's average QPS is appended to the models' history and a prediction is made for the next minute's average QPS based on the historical values. The AI models work in a racing environment: before predicting the QPS for the next minute, each model's accuracy is calculated using (12), and the best performing one is used, as depicted in V-C.

The predicted metrics and, in case of a fallback, CPU metrics are reported from the Connector to HPA as custom metrics. The vanilla HPA module then makes the scaling decision based on these metrics.

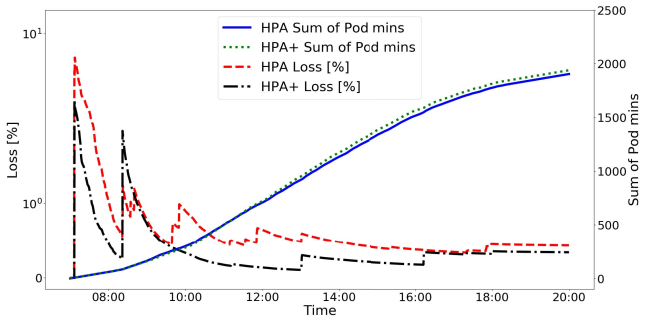
B. Adaptive operation under excess policies

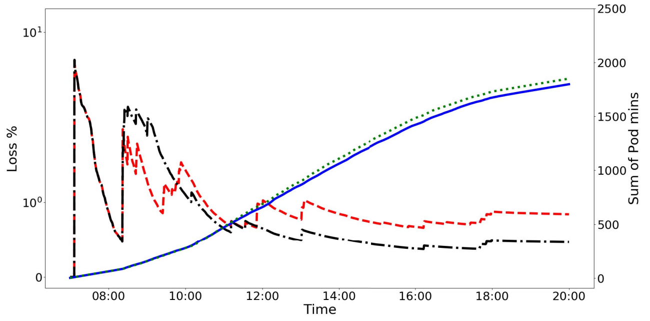
An application provider that uses Kubernetes or a similar management system to run its application pays for the cloud resources the system reserves, while it provides QoS to its users according to the SLA. The application provider may have various strategies depending on the cost of running the application 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, and determine when the application provider requires scaling event. 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% of capacity. 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 strategy 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.





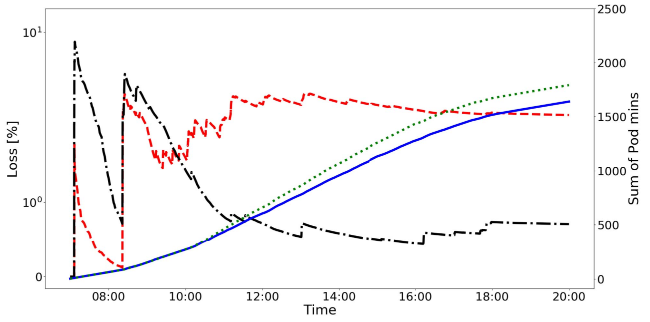


Fig. 8. Comparison of HPA and HPA+ with Conservative (top), Normal (center), Best effort (bottom) scaling strategies

C. *Results of HPA+ experiments*

We tested the implemented HPA+ system, described in Subsec. VI-A, in a real life scenario with the different scaling strategies and compared the results to the native HPA. 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 two metrics. In Fig. 8 the moving percentage of lost requests (left y-axis) and the 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 costs $1.

As shown in Fig. 8, 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 experience. Even at its highest level, with the Best effort scaling policy, HPA+ leads to a drastically lower number of lost requests compared to HPA.

Policy	Lost requests	Resource usage
Conservative	-22%	+1.8%
Normal	-44%	+3%
Best effort	-72%	+9%

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

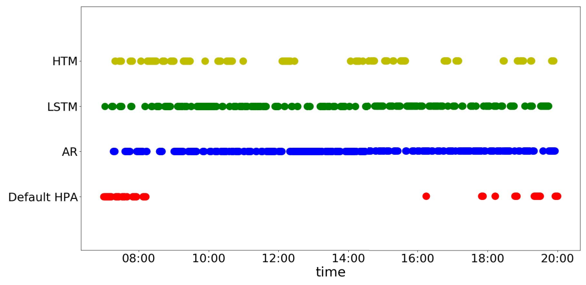


Fig. 9. Activated forecast/scaling methods during the day

The final comparative results of HPA and HPA+ operations can be seen in Table I. In the first column the three scaling policies are listed. The second column shows the reduction in loss of 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 four scaling methods (CPU-based HPA, AR, LSTM and HTM predictions), the scaling engine strives to use the one which suits best the actual traffic. In Fig. 9 the four scaling methods are shown with the points in time when they were active under the Normal scaling policy. At the beginning and 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 that particular measurement AR ran 44%, LSTM 32%, HTM 16% and default HPA 8% of the time.

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 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. 10 the Sum

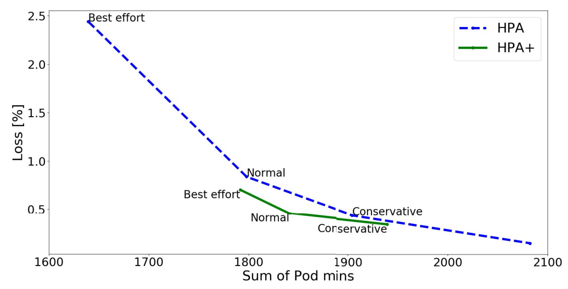


Fig. 10. Sum of Pod mins and loss for various scaling policies

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.

VII. Conclusion

The concept of microservices and the 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 and cloud operators. Application providers provisioning the novel services over cloud platforms need appropriate and easy to use configuration interfaces to control different aspects of 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 and an analytic model was proposed to capture the main characteristics of the broadly used method. The model can play important role for example in the resource 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 AI-based forecast methods in order to be able to optimize the operation in different circumstances. The proposed solutions were evaluated via simulations and real measurements where the input traffic was constructed from real traces collected from a campus network. We also focused on easy configurability, and introduced a simple management parameter, called excess parameter, which is capable of controlling the resource allocation policy by specifying the targeted compromise between resource over-provisioning and SLA violation.

We found that our enhanced auto-scaling engine is able to significantly decrease the number of rejected requests, of course, at the cost of slightly more resource usage. However, the additional resource consumption is negligible compared to the regular operation. Furthermore, by setting the excess parameter, the single management turnkey parameter, so that HPA and HPA+ consumed similar amount of resources, our enhanced auto-scaling engine provided a lower lost request ratio. As a future work, we plan to add more precise and sophisticated prediction models to our HPA+ solution, which can further improve the overall performance of the system.

References

[1] Fortio. http://fortio.org/.
[2] HAProxy. https://www.haproxy.com/.
[3] Horizontal Pod Autoscaler - Kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[4] kops - Kubernetes Operations. https://github.com/kubernetes/kops.
[5] NASA-HTTP logs. ftp://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.
[6] Prometheus - monitoring system & time series database. https://prometheus.io/.
[7] WorldCup98 HTTP logs. ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html.
[8] Al-Dhuraibi et al. Autonomic vertical elasticity of docker containers with elasticdocker. IEEE CLOUD, 2017.
[9] Al-Dhuraibi et al. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2017.
[10] Arabnejad et al. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. IEEE/ACM CCGRID, 2017.
[11] Barrett et al. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.
[12] Catalão et al. Short-term electricity prices forecasting in a competitive market: A neural network approach. *Electric Power Systems Research*, 77(10):1297–1304, 2007.
[13] Chen et al. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *USENIX NSDI*, 2008.
[14] Dutreilh et al. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS*, 2011.
[15] Fabiana et al. Horizontal and vertical scaling of container-based applications using reinforcement learning. IEEE CLOUD, 2019.
[16] Gourav et al. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. IEEE CLOUD, 2019.
[17] Jeff Hawkins and Sandra Blakeslee. *On intelligence: How a new understanding of the brain will lead to the creation of truly intelligent machines*. Macmillan, 2007.
[18] Horovitz et al. Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning. IEEE FICLOUD, 2018.
[19] Islam et al. Empirical prediction models for adaptive resource provisioning in the cloud. *Elsevier Future Generation Computer Systems*, 28(1):155–162, 2012.
[20] Jia et al. MMPP/M/C queue with congestion-based staffing policy and applications in operations of steel industry. *Springer Journal of Iron and Steel Research International*, 26(7):659—-668, 2018.
[21] M.A. Kaboudan. A dynamic-server queuing simulation. *Computers & Operations Research*, 25(6):431 – 439, 1998.
[22] Hamzeh Khazaei et al. Elascale: Autoscaling and monitoring as a service. In *CASCON*, 2017.
[23] Li et al. Using neural networks to estimate wind turbine power generation. *IEEE Trans. Energy Convers.*, 16(3):276–282, 2001.
[24] Lorido-Botran et al. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.
[25] Mazalov et al. Queuing system with on-demand number of servers. *Mathematica Applicanda*, 40(2):1–12, 2012.
[26] Messias et al. Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. *Springer Neural Computing & Applications*, 27(8):2383–2406, 2016.
[27] Jeremy Miles. R squared, adjusted r squared. *Wiley StatsRef: Statistics Reference Online*, 2014.
[28] Rao et al. Vconf: a reinforcement learning approach to virtual machines auto-configuration. ACM ICAC, 2009.
[29] Wen et al. Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones. IEEE INFOCOM, 2012.
[30] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16, Oct 2010.