e-Informatica Software Engineering Journal, Volume 16, Issue 1, 2022, pages: 220107, DOI: 10.37190/e-Inf220107

Microservice-Oriented Workload Prediction Using Deep Learning

Sebastian Ştefan*, Virginia Niculescu*
*Faculty of Mathematics and Computer Science, Babeş-Bolyai University
stefansebii@gmail.com, virginia.niculescu@ubbcluj.ro

Abstract

Background: Service oriented architectures are becoming increasingly popular due to their flexibility and scalability which makes them a good fit for cloud deployments.
Aim: This research aims to study how an efficient workload prediction mechanism for a practical proactive scaler, could be provided. Such a prediction mechanism is necessary since in order to fully take advantage of on-demand resources and reduce manual tuning, an auto-scaling, preferable predictive, approach is required, which means increasing or decreasing the number of deployed services according to the incoming workloads.
Method: In order to achieve the goal, a workload prediction methodology that takes into account microservice concerns is proposed. Since, this should be based on a performant model for prediction, several deep learning algorithms were chosen to be analysed against the classical approaches from the recent research. Experiments have been conducted in order to identify the most appropriate prediction model.
Results: The analysis emphasises very good results obtained using the MLP (MultiLayer Perceptron) model, which are better than those obtained with classical time series approaches, with a reduction of the mean error prediction of 49%, when using as data, two Wikipedia traces for 12 days and with two different time windows: 10 and 15 min.
Conclusion: The tests and the comparison analysis lead to the conclusion that considering the accuracy, but also the computational overhead and the time duration for prediction, MLP model qualifies as a reliable foundation for the development of proactive microservice scaler applications.

Keywords: microservices, web-services, workload-prediction, performance-modeling, microservice-applications, microservice scaler

1. Introduction

Microservice architectures are considered to be the next step in the evolution of Service Oriented Architectures (SOA) that were popularised in the 90s [1]. Some particular aspects of the microservices are their fine granularity, focus on decoupling, scalability, usage of lightweight protocols, and strong DevOps integration [2]. They are currently seeing a huge adoption rate: a survey of Kong Inc. done in the summer of 2019 with 200 technology leaders at large U.S. companies has revealed that 84% of them have embraced microservices, and 40% believe that organizations will fail within 3 years if they do not keep up with these [3]. Furthermore, microservices are a good fit for cloud deployments, proven by the

Submitted: 17 Sep. 2021; Revised: 18 Feb. 2022; Accepted: 09 Mar. 2022; Available online: 25 Mar. 2022

large scale operations of companies like Amazon, Netflix and LinkedIn, and their reported improvements after switching from the monolithic model [4].

Workload prediction is important in order to ensure efficient scaling of these services and optimisation of cloud resource usage, which means starting up new services during periods of high traffic and stopping some of them when resources are not needed. An analysis of the application of microservices, described in [4], has shown that the use of tools designed to deploy and scale microservices reduces infrastructure costs by 70% or more. Improving workload prediction performance means equipping scaler services with better tools for dealing with unexpected traffic spikes, which is translated in both a smoother experience for users and lower costs for maintainers. Autoscaling is the most practical solution since it assures the automatic scaling of microservice instances in order to meet the SLA(Service Level Agreement) [5], without a human agent analyzing and constantly taking scaling decisions. It can be *reactive* or *predictive*, the latter considering multiple inputs like historical information and current trends, in order to predict future traffic patterns.

The main goal of the presented investigation was to find a performant model for microservices workload prediction, which can be later used by a proactive microservice scaler. As a consequence of our goal, the research question that led our investigation was:

"*Do deep learning algorithms lead to better results than classical time series approaches for workload predicting of a microservice autoscaler? If yes, which one is the most appropriate?*"

Previous research in this field mainly uses classical time series approaches (such as ARIMA – autoregressive integrated moving average, Brown's quadratic exponential smoothing or WMA-weighted moving average) [6–8], or simple machine learning [9, 10]. Our investigation uses different deep learning architectures: MLP (Multilayer Perceptron), CNN (Convolutional Neural Network), hybrid CNN-LSTM (CNN Long Short-Term Memory Networks); deep learning was shown to outperform classical methods on some time series prediction tasks [11], and we selected some models of varied complexity.

The contribution of this research is twofold:

– A microservice-oriented prediction methodology adapted to the particularities of this setting, is proposed. The methodology includes steps and decisions that were taken to match practical microservice demands, such as choosing to predict the number of requests, which is a metric that is not influenced by the scaling prediction, and making the prediction in time intervals of an order of minutes and predict a step into the future to allow time for services to be deployed to match the expected traffic. The prediction window size was chosen for accuracy while also allowing time for most application servers or containers to initialize the application. This methodology is also covering data preparation and processing, that is designed for prediction accuracy.
– A comparative analysis of the performance of different prediction models inside the proposed methodology is conducted; the comparison is done between the results obtained using the chosen deep-learning algorithms, and classical time series approaches, but also with some hybrid machine learning models used in industry [9]. The comparison shows important improvements over the previous results, and emphasizes MLP as the best choice for a predictive microservice scaler. MLP seems to be the most appropriate to capture the complexities of the dataset while also having the advantage of faster training time.

The paper is structured as follows: After we present the related work in the next section, we succinctly describe microservice characteristics and the practical aspects which influenced the lines of this research in Section 3.1. Section 3 introduces the proposed methodology and in Section 4 we refine it in substeps and specify the settings for our experiments. Section 5 presents our practical implementation of the methodology on a specific dataset: the baseline models' results, the tuning process of the deep learning models for selecting the hyperparameters, an evaluation of the best performing ones, and a comparison with the baselines and other research work. Section 5.4 summarises the obtained results, and in addition, in order to emphasise their utility, a proof of concept implementation for an auto-scaling tool using this model is presented. Conclusions and future work are presented in Section 6.

The following abbreviations are used in the paper: ANN (Artificial Neural Networks), ARIMA (AutoRegressive Integrated Moving Average), CNN (Convolutional Neural Network), CNN-LSTM (CNN Long Short-Term Memory Networks); FFT (Fast Fourier Transform); MAE (Mean Absolute Error); MAPE (Mean Absolute Percentage Error); MRE (Mean Relative Error); MLP (Multilayer Perceptron); MSE (Mean Squared Error); RMSE (Root Mean Square Error); RSLR (Robust Stepwise Linear Regression); SVM (Support Vector Machine); VM (Virtual Machine).

2. Related work

Different classical time series models have been applied for web-services workload prediction. Calheiros et al. [6] apply the ARIMA model to cloud workload prediction. The model was evaluated using a trace of English Wikipedia resource requests spanning a duration of four weeks. The data of the first three weeks are used for training and the fourth for prediction using a time window of 1 hour. The obtained MAPE varies from 9% to 22% depending on the confidence interval, which was chosen from 80 to 95 in order to limit the occurrence of underestimations.

Other classical time series models have also been applied, like Brown Exponential Smoothing by Mi et al. [7] obtaining a MRE of 0.064 on the France World Cup 1998 web server trace. Another classical model is Weighted Moving Average, in which recent observations receive more weight than older ones, was applied by Aslanpour et al. [8], and was tested on a NASA server 24h trace, achieving a 5% improvement in response time on a cloud scaling simulator.

It is difficult to identify the best of these classical approaches for our task since the research outlined above used different datasets and evaluations measures. However, we can look for comparisons between different classical models on other time series problems (not necessarily related to workload prediction). Udom and Phumchusri [12] show that ARIMA performs better than other models (Moving Average, Holt's and Winter's exponential methods) in terms of MAPE on four different datasets. ARIMA was also shown to perform better on a short-term forecasting dataset than an exponential smoothing approach [13]. Zhu et al. [14] show that ARIMA outperforms Holt's exponential smoothing model in terms of MSE on air quality time series analysis.

Khan et al. [15] have used Hidden Markov Models to predict workloads for a cluster of VMs. The used dataset comes from an in-production private cloud environment, and the selected metric is the CPU utilization of the VMs. Their model identifies VMs which have similar loads, trained on a trace of 17 days and generates predictions for intervals of 15 min for the next 4 days. Still, their approach only works for a static configuration, because the training dataset is a matrix of the all VMs in the system on the all selected time intervals, and the selected metric is the CPU utilization. This means that if the configuration of the system changes then the accuracy will not be preserved, and utilization data for the new VMs must be built, and then a retraining session is necessary. We try to propose a model which can dynamically adapt to scaling decisions without penalty in prediction accuracy.

Another example of a static system predictor is the one proposed by Syer et al. [16], which detects variation in workloads between test and production environments for multiple large-scale software systems from the telecommunications domain. As opposed to this approach which discovers various types of workloads and their deviation from the training environment, but can not adapt automatically to system re-configuration, our solution assumes requests homogeneity (discussed in Section 3) and can adapt to automatic scaling events.

Kumar and Singh [10] applied ANN for workload prediction on a seven month log of traffic from a Saskatchewan University web server and a two month one from the NASA Kennedy Space Center web server. They use a classical ANN architecture: one input layer (size 10), one hidden and one output layer, and the model is trained through the SaDE technique, which means learning the weights through evolutionary algorithms. The results of this model were compared to an ANN trained through backpropagation. The model trained with SaDE got 0.013 and 0.001 RMSE on the selected data sets, while for the one with backpropagation a RMSE of 0.265 and 0.119 was obtained.

CloudInsight [9] is one of the most complex models for workload prediction. It uses a technique called "council of experts" – an ensemble of different models, which in this case are: classical time series (autoregressive, moving average, exponential smoothing), linear regression, and machine learning – SVM. Each model has a different prediction weight, which is also real-time learned through a SVM, based on their accuracy on the dataset. The evaluation was done on a subset of the Wikipedia trace [17], on Google cloud data, and on some generated workloads. They indicated that ARIMA and SVM are the two best static predictors they have experimented with. Considering as a performance indicator the normalized RMSE, on average, the ensemble system was 13%–27% better than the baselines (ARIMA, FFT, SVM, RSLR).

A review of how deep learning methods can be applied to time series problems was presented by Gamboa in [11]. The paper distinguishes between three types of problems: classification, forecasting and anomaly detection, presents methods for modeling them, and guidance for selecting appropriate models. It also shows that using these, an improvement in performance could be achieved, on case studies for different applications in which deep learning performed better. Brownlee [18] published a comprehensive guide on applying MLPs, CNNs and LSTMs on various real datasets, and discussed their advantages over classical methods, which were used as baselines for the experiments. The study highlighted the ability of deep learning models to find non linear relationships in data, as opposed to linear methods, like ARIMA; this was the reason to focus on this kind of methods in our investigation.

Lin et al. [19] proposed a hybrid CNN-LSTM architecture for learning trends in time series. It relies on CNN to extract important features from raw time series data, and passes them to the LSTM layers to find long range dependencies in historical data. The model was shown to outperform both CNN and LSTM with around 30% lower RMSE on three real world datasets. These results look promising, and for this reason this is one of the models taken into consideration for our experiments.

There are some approaches for workload prediction of large scale systems that use LSTM models such as Tang et al. [20], Zhu et al. [21] which show it to be a suitable approach. In our experiments, we have tested the hybrid CNN-LSTM model with the expectation that it would perform better than its individual components.

Zhang et al. [22] used deep learning based on canonical polyadic decomposition to predict workloads for cloud applications (in this case using a trace of 10 days for the PlanetLab platform, which is a global research network that supported the creation of new network services [23]). Their results indicate better performance of the deep learning model than of the state-of-the-art machine learning based approaches. However, while the model is robust in terms of request workload variety, it aims to predict CPU utilization, which, as outlined above, is not a good fit for our investigation.

A significant description of the necessity and of an implementation of a predictive autoscaler for microservices was done by Netflix in [24]. Before implementing *Scryer*, the name of the aforementioned service, they relied on Amazon Auto Scaling service of the Amazon Cloud, which was based on a reactive approach. *Scryer* uses classical time series methods such as Fast Fourier Transformation, which models a sinusoidal over the input data, and linear regression on clusters of points from the predicted time window in previous days. This model addressed three problems encountered with Amazon's scaler: dealing with rapid spikes in demand by preparing ahead of time, restoring compute capacity after outages, and factoring known usage traffic patterns. Netflix are one of the pioneers of microservice technologies, having broken down their monolith application into multiple services covering everything from video streaming, account registration, content recommendations, in the early 2010s, and later becoming an authority in this domain by developing a strong presence in the open source community based on publishing their tools [25].

A predictive scaling policy was later added in Amazon Web Services [26], based on machine learning algorithms. However, this feature is not yet available in other cloud providers such as Microsoft Azure [27] or Google Cloud [28].

Building on top of the related work presented in this section, we aim to apply and compare some deep learning methods, which were shown to be suited for time series in [18] and [19], for the specific task of workload prediction. The success of this task is highlighted by comparing error metrics with those reported by CloudInsight [9] – the specified ensemble of classical and machine learning approaches, on the same dataset, which is a subset of Wikipedia traces.

3. Scaler prediction methodology

This section presents some of the most important characteristics of Microservice architectures in the first part. Based on these characteristics, in the second part we present our proposed scaler prediction methodology which can be applied to any particular implementation of this architecture.

3.1. Microservice characteristics

Web services are generally associated with Service Oriented Architectures (SOA) [1]. The main idea of this type of architecture is to break down monolithic applications into independent parts that are loosely coupled, autonomous, offer a standard contract and act mostly as black boxes to their consumers. This means that services can be developed, updated and deployed independently offering better scalability than traditional architectures.

Microservices [29] are the modern approach of Service Oriented Architectures, and they have several important characteristics [2]:

1. **Fine Granularity** – each service is implemented to serve a specific business case.
2. **Maintainability** – changes of a feature will have limited impact on the overall code-base.
3. **Reusability** – you can select which features to import into a different system.
4. **Agility** – bug fixes and new features can be deployed without retesting or taking down other parts of the system.
5. **Autonomy** – they are separate entities, with their own tech stack, and can be deployed independently.
6. **Loose Coupling** – they communicate using lightweight network protocols such as REST and HTTP.
7. **High Scalability** – due to their autonomy and loose coupling they can scale-out horizontally without incurring heavy communication overhead.

3.1.1. Deployment

All these characteristics make microservices a good fit for cloud deployments. Cloud providers generally offer on-demand resources, which is more convenient for hosting the applications, and allowing less expensive dynamic workloads [4]. If application workloads are fluctuating, then it is advisable to scale the services accordingly, in order to provide smooth experience for the users, and in the same time to use the resources efficiently.

The problem of having unused resources during the periods of low traffic is solved automatically by cloud deployments, by allocating them to some other users who need them. Similarly, it may be necessary to request more resources when a traffic spike is foreseen. Microservice architectures are ideal for these operations because they offer high level of scalability. Due to their fine granularity it is possible to scale only the services that are in high demand, which would not be possible on monolithic applications. Also, since they are designed to be autonomous, it is simple to setup necessary dependencies such as databases without conflicts among instances.

Also, service discovery is one of the key tenets of a microservice-based architecture. Trying to hand-configure each client or to define some form of convention can be very difficult and also unsafe. In order to overcome these kinds of problems service discovery applications are offered. For example, Eureka is the Netflix Service Discovery Server and Client [30]; this server can be configured and deployed to be highly available.

3.1.2. Scaling

The microservices could be scaled manually, which is inefficient, or automatically through a dedicated service. Autoscaling is the process of automatically scaling out instances in order to meet the SLA(Service Level Agreement) [5], which is formed of a list of commitments between clients and service providers, related to different aspects of the service, such as: quality, availability, responsibilities. For example, it could be stated that the application should have 99% uptime, or it should respond to most requests within a given time range.

Autoscaling can be *reactive*, by setting up thresholds such as resource utilization, and instantiating new services when they are reached, or *predictive* by creating new instances ahead of the foreseen traffic spikes. Predictive autoscaling considers multiple inputs like historical information and current trends in order to predict future traffic patterns.

Even though predictive auto-scaling can be done efficiently without having a cloud deployment, for example scaling some service up and other down alternatively on a fixed resource environment, cloud environments are the best fit and research work is done to address this need [31].

3.2. Proposed scaler prediction methodology

We propose a methodology for finding a prediction model to be used by a proactive scaler for microservice architectures, which takes into account the specific characteristics outlined in the previous section. The main steps are the following:

- **Choose one type of services for which to define a proactive scaler**
  - Each microservice type should have its own predictive autoscaler; a microservice is specialized for a single specific task and it will operate very specific requests.
  - It is expected to obtain better prediction accuracy and resource utilization when working with a single type of requests. The request type would increase the dimensionality of the input data, therefore increasing computational resource utilization, and finally impact on the prediction accuracy, since the model would have to learn multiple features (the error will increase proportionally to the number of the request types). Additionally, to put it into practice, a new model would be required in order to estimate how many resources need to be allocated based on multiple request counters, but also on interactions between them.
- **Choose the number of requests per resource to be the selected metric for prediction**
  - The reason for choosing this metric is this metric independence of the scaler's action. Metrics such as CPU utilization or response time, predicted in [32], are affected by the outcome of the predictor, making them an unreliable target. Also, this is in line with the research done by Jindal et al. [33], who proposed a metric for measuring microservice performance based on the number of satisfied requests, called MSC (Microservice Capacity). Thus, a proactive scaler can determine the number of required instances by dividing the predicted incoming traffic to the MSC.
  - Microservices have fine granularity, therefore we can assume request homogeneity – the requests for one specific microservice are uniform (i.e., they could be solved in a similar period of time). This means that for this problem we can use this simple metric without compromising the usefulness of our predictions.
- **Model real service trace data analysis as a time series supervised learning problem**
  - A common dataset which can be extracted from any application's log is a list of timestamps when requests were handled (one such dataset could be extracted from each microservice type, as they are highly autonomous and we can demarcate exactly the requests they received and when they were completed).
  - It is possible to extract more useful information from this data if we model it as a time series problem [34]. Since specific timestamps are not required, but just general access patterns, a feasibly approach would be to group requests into a series of *buckets* (abstraction used for representing time series). A bucket has a fixed width (some time range) and variable height (the number of requests handled by the program in that range). A visualization of such a time series model is presented in Figure 1.

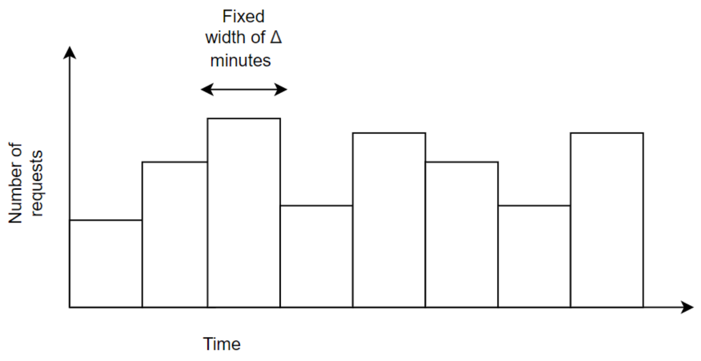

Figure 1. Abstract representation of a time series modeling traffic of one microservice type

  - Microservices are highly scalable, so we need to achieve a granularity in predictions as fine as possible, meaning that scaling decisions can be taken as soon as the information is available. We can control this granularity in terms of selecting the width of the time series buckets. Therefore the lower bound (which we are aiming for) of the bucket width is dependent on technological constraints for scaling up/down microservices. This estimation also fits an observation from [15] which has used a 15 min bucket width, from analyzing autocorrelations on their dataset, which consists of a 21 days trace from an in-production distributed application.
  - After converting the dataset to time series it must be prepared for being fed to a supervised learning algorithm (the supervised learning is considered because we already know the desired prediction target and we can label our data [35]) which means transforming the series into a list of vectors of the form $(input, output)$. A possible choice for this transformation is based on *sliding window* technique (which was shown to have adequate performance and allow for a wide range of algorithms to be applied to the resulting dataset [36]); more details about using this process of data preparation is detailed in Section 4.2.
  - Also, we are not interested to predict the height of the first next bucket in the future, because the scaling decisions might be useless if they can not be executed in practice, meaning that a time is required between the moment in which the scaling decision is taken and the moment when the new microservice application instance is online and can actually process requests. This period of time was outlined previously as the ideal width of a bucket. In order to accommodate this requirement we need a classical approach for multi-step time series prediction (e.g., the Direct strategy from [37]), in which the prediction target is the second window in the future.
  - The prediction window is limited to one, in the near future, in order to improve accuracy. This requires periodic predictions, however, once a deep learning model is trained, the actual computational overhead is small (less than a second in our experiments).
- **Apply an appropriate prediction model**
  - Choosing the most appropriate model is a complex problem, and our empirical investigation aimed to provide such a model.
- **Evaluate the results**
  - estimate the prediction error using different metrics;
  - compare the results with similar results obtained using with different prediction models;

– verify using practical usage.

The summary of the proposed methodology is depicted using a diagram in Figure 2.

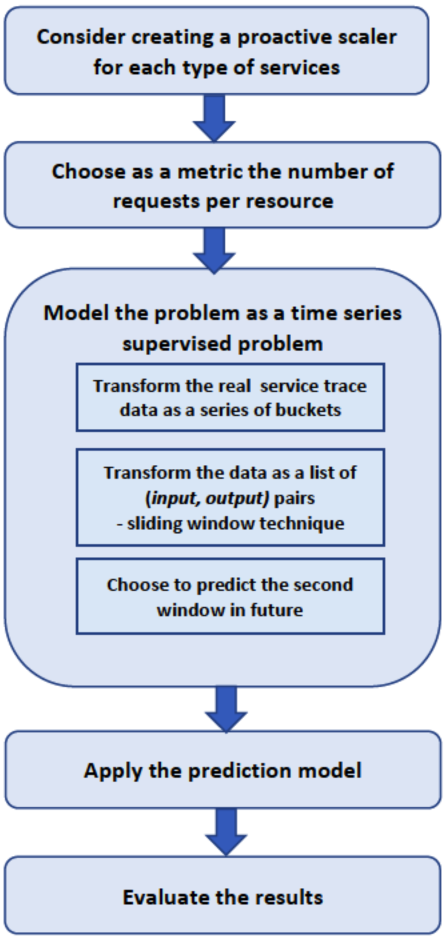
Consider creating a proactive scaler for each type of services
Choose as a metric the number of requests per resource
Model the problem as a time series supervised problem
Transform the real service trace data as a series of buckets
Transform the data as a list of *(input, output)* pairs - sliding window technique
Choose to predict the second window in future
Apply the prediction model
Evaluate the results

Figure 2. Diagram associated to the methodology for finding a prediction model for a proactive microservices scaler

In order to estimate the most appropriate prediction model for microservice workload prediction (to be applied in the fourth step), we have conducted an investigation based on the specified methodology. This investigation started by choosing and preparing data on which we can do the experiments, extracting a collection of prediction models that are potential candidates, followed by the preparation of their initial settings. After that we did the experiments and the evaluation of the results.

4. Methodology refinement and investigation settings

In this section we present details regarding the data used in the experiments, their preparation as corresponding supervised datasets, and the collection of prediction models that we have chosen as potential candidates.

4.1. Data sets

Very important aspects in the selection of the datasets were to follow real world user traffic patterns, to have a consistent size, and to have some variation which would showcase how the model can handle unpredictable spikes. Based on these we have chosen several Wikipedia traces. Although we do not know the specific implementation of the Wikipedia server, this dataset can be used for testing the model for two reasons:
– the requests are all of the same type (fetch the content of a wiki page) which is in line with the assumption of request homogeneity for microservices, and
– the traffic patterns come from a production server and capture realistic user traffic (random spikes, day/night variation, weekend variations, etc.).

In addition, the appropriateness of this choice is also confirmed by the fact that similar datasets were used in the analysis conducted by Kim et al. [9] that describes the algorithms for the CloudInsight service, which is a commercial cloud scaling and monitoring platform.

The raw data used for the experiments is a Wikipedia trace for 12 days in September 2007 [17], available online at http://www.wikibench.eu/. From this, two subsets of requests were extracted as separate datasets: all requests to Japanese and German Wikipedia, respectively, to facilitate the results comparison with those obtained by Kim et al. [9] which were based on the same data.

The Japanese wiki dataset is presented in Figure 3. The y-axis represents the number of requests and the x-axis the time, measured in 10 minute intervals over the whole period. It shows an interesting variation in the form of a large spike during the 5th day of measured data which could be a challenge for some prediction models.

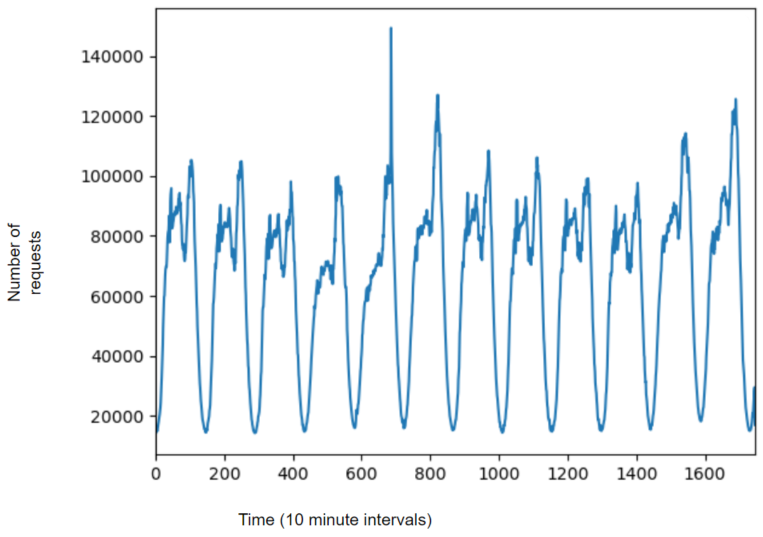

Figure 3. Japanese Wikipedia data visualization: number of requests per 10 minute intervals

The first dataset contains 111 million requests (ja.wikipedia) and the second 101 millions requests (de.wikipedia). The amount of data after preparation is the same as if we would extract from the global trace during the same time range, but with a much faster processing, because the number of buckets depends on the considered time interval, and not on the number of requests.

Since the target bucket width is given by the time in which a microservice application can be reasonably started up we can do some estimations about general technical constraints of this operation. For example, if we considered the Netflix open source stack that is among the most popular approaches for implementing microservices, we have to consider the time for initializing Spring Boot, service discovery (e.g., Eureka [30] – the Netflix default client – needs a refresh time of 30 s, which is recommended on production environments, too), and in some cases performing business logic like initializing in-memory caches from database information. The typical initialization time for microservice frameworks will also add a few seconds [38]. In addition, a typical deployment may also need time for starting up the container (e.g., Docker) or virtual machine. Considering that there are many factors which can influence this interval in our experiments we considered a permissive estimation of a few minutes. Therefore we have chosen two cases for target bucket width: 10 min and 15 min.

4.2. Data preparation

In order to turn a web request log file into a supervised dataset the following steps were taken:
- extract timestamps of all requests for a country (e.g., all lines matching ja.wikipedia);
- create buckets that contain the number of requests in a time interval;
- iterate over the buckets using the sliding window technique, and group them into (input, output) tuples.

Applying sliding window. The starting point for the sliding window time series technique [39] is a time series $(t_1, t_2, \ldots, t_{size})$, where t_i is the number of requests in the i-th bucket. Training instances are then generated with input $(t_i, t_{i+1}, \ldots, t_{i+n-1})$ and output (t_{i+n+1}), where n is the size of the sliding window. This process starts at $i = 1$ and is incremented by 1 until $i = size - n + 1$. The predicted value is t_{i+n+1} instead of t_{i+n} because a scaler using this model would need to have a buffer window during which to deploy the services. These are emphasized in Figure 4.

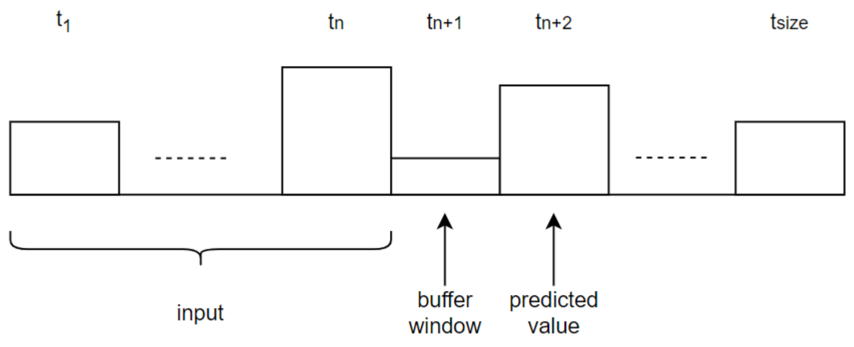

Figure 4. Sliding window technique

Input data were scaled using the min-max scaling technique: $x = \dfrac{(x - \min)}{(\max - \min)}$, which brings the dataset into the $[0, 1]$ range. The same method was applied by Kumar and Singh in [10] in order to speed-up learning. In a practical implementation, this scaling step is more difficult to apply because it should rely on some hypothetical bounds that have to be determined for future traffic. Still, these bounds could be estimated based on the historical data.

The sizes of the datasets, after applying transformations were 1166 for the 15 min window and 1747 for the 10 min window. The sizes are determined by the sampling window of 12 days and the bucket windows of 10 and 15 minutes.

Performance metrics

The error metrics selected in this investigation are:

$$\text{Mean squared error: } MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2,$$
$$\text{Mean absolute error: } MAE = \frac{1}{n}\textstyle\sum_{i=1}^{n}|Y_i - \hat{Y}_i|, \text{and}$$
$$\text{Mean absolute percentage error: } MAPE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)/Y_i,$$

where Y_i and $\hat{Y}_i$ are the observed, respectively, the predicted values [40]. MSE was used as the loss function for training because it tends to penalize big deviations in prediction, which is desirable for our problem as we want to accurately predict traffic spikes. MAE is similar, but conceptually simpler, given that each prediction error contributes in proportion to its absolute value. MAPE is independent of the problem scale and can be interpreted intuitively, therefore can be used to give a general evaluation of how well a model performs across different datasets. According to Lewis [41] a highly accurate forecast would have MAPE lower than 10%, and a good forecast between 10% and 20%.

4.3. Baseline models

Baseline models were considered in order to verify in which measure machine learning is useful for this problem, and if using it, features not considered by simpler methods could be learned. Two baseline models were applied: a naive approach, and a classical time series model – ARIMA.

The naive approach just assumes that the predicted workload is the same as the last observed workload. No proactive scaler could use this model as the predicted change in traffic is always null, but it is used in order to check if the proposed models perform better than doing no prediction at all.

ARIMA [42] is a classical approach for modeling time series. It has been selected because it has been applied with good results to workload prediction before [6], and was shown to perform better than other classical models [12–14]. Also, it is a common baseline model for machine learning solutions in time series predictions [9, 43–45]. Furthermore, it combines multiple simpler models (AR and MA) into a performant one. Autoregressive models (AR) make predictions based on previous observations while Moving average(MA) models use recent forecast errors. The integrated part indicates whether the series needs to be differenced, and how many times. Therefore, the parameters of the ARIMA model are:
– *p*: the number of lag observations included in the model;
– *d*: the number of times that the raw observations are differenced;
– *q*: the size of the moving average window.

4.4. Deep learning models

We have chosen in this investigation the following deep learning architectures: MLP, CNN, CNN-LSTM hybrid, since all have been shown to perform well on time series tasks [11, 18, 19]. Also, deep learning has constantly outperformed classical methods in prediction tasks [46].

All the selected models have advantages and drawbacks among each other regarding training speed, the amount of data required to produce good answers, and the tuning of the size of the sliding window to capture relevant recent information.

4.4.1. MLP – Multilayer perceptron

MLPs are quintessential deep learning models that although efficient in their own right also serve as baselines for more sophisticated architectures [47]. It is made up of an input layer, a number of hidden layers and an output layer, linked by weights which are learned through the backpropagation algorithm. While a MLP with one hidden layer is theoretically sufficient to represent any function, that layer may be too large and training could be affected by overfitting, therefore deeper models can help reducing the generalization error [48].

This model has been selected to check whether looking at a smaller sliding window, without taking into account further historical dependencies, achieves satisfactory results.

4.4.2. CNN – Convolutional neural network

CNNs are specialized in dealing with data that has a grid-like topology such as images (2d) or time series (1d) [47]. They have the ability to learn filters which assign importance to some aspects of the input data, which is done by the convolutional layers. Another type of layers that they usually contain are the pooling layers, which reduce the spatial size of the convolved features and make the representation invariant to small translations in input.

CNN has been tested in this experiment because it looks like a natural fit for a time series problem given its assumed spatial dependencies. CNNs can extract only the important features of the input, therefore they can efficiently work with a larger sliding window, and take into account more recent measurements when making a prediction.

4.4.3. CNN-LSTM – CNN long short-term memory networks

The CNN-LSTM architecture involves using Convolutional Neural Network (CNN) layers for feature extraction on input data combined with LSTMs to support sequence prediction. This hybrid model was applied on a range of time series tasks by Lin et al. [19] and was shown to outperform both CNN and LSTM models.

Recurrent Neural Networks (RNN) are Neural Networks that take into account the outcome of previous predictions, while making the current one [47]. LSTM – Long Short-Term Memory, networks are an improvement over RNNs in the sense that they are better at capturing long-term dependencies [49].

This model has been chosen because it combines the ability of CNNs to extract salient features from raw time series data with the capability of LSTMs to find long range dependencies and historical trends.

5. Experiments and evaluation

In this section we describe the experiments that we conducted, their results, and a comparative analysis of these results. The research follows a set of best practices such as: setting baselines, starting with parameters that have been shown to perform well on other problems, exploring possible solutions manually and automatic exhaustive search for fine tuning parameters. The experiments were performed using the data and models described in Section 4.

First a tuning phase has been carried out for each chosen architecture in order to choose the best parameters for each model. The details of this phase are presented in the next subsection.
For the validation we have used k-fold validation [50], which estimates how well a model will perform on previously unseen data and offers a less biased skill estimation than the classical train/test validation method. The k-fold Cross-Validation with $k = 3$ was chosen, which means splitting the training dataset into $k = 3$ equal parts; for cross-validation $k - 1$ parts are used to perform the training (the weights are reinitialised for training on each subset), and the evaluation is done on the part left out, and this process is repeated until an evaluation was done on each of the parts. Finally, the averaged accuracy of all tests was considered. The instances themselves were not shuffled inside the partitions, as their ordering is significant for LSTM models.

Each dataset was split into training (the first 90% of data points) and testing (the remaining 10%) data. After tuning (on the training set of a selected dataset), the resulting models were next trained again on all training datasets, and evaluated on the testing data, which were unseen during tuning and training.

Implementation. All the selected models were implemented in Python programming language. For machine learning models the Keras library [51] was used with some variations (described below) on the following types of layers: Dense for MLP, Conv1D, MaxPooling1D and Dense for CNN, and the previous ones with the addition of LSTM for CNN-LSTM hybrid.

The k-fold validation process was carried out using the scikit-learn library [52]. Statsmodels library [53] was used for ARIMA implementation.

Aside from the configurations described in the article, the default settings of the library were used. The algorithm used in initializing the connection weights of our neural networks models was Glorot Uniform provided by Keras, also called the Xavier initializer [54].

5.1. Hyperparameter optimisation

We have chosen for the tuning phase the Japanese wiki dataset (on the first 90% data points) described in section 4.1 because, besides the fact that includes significant patterns, it also has some interesting irregularities, like a huge spike which is not repeated. As we have previously mentioned, this dataset was also used by Kim et al. [9] and we intend to compare the results.

The selected time window for tuning was set to 10 min, because this is a reasonable prediction time to allow a scaler to spin out new instances, as shown in some previous experiments [24].

5.1.1. Naive baseline

The naive baseline leads to the following results: 2.02×10^7 MSE, 3577.8 MAE and 7.1% MAPE. This illustrates the fact that although the MAPE score would classify it as a very good predictor, it does not do anything useful and the proposed models should achieve better results.

5.1.2. ARIMA

Settings. In order to apply the ARIMA model we had to find appropriate values for its parameters: p, d, q. The value of d represents the number of times the series needs to be differenced in order to make it stationary. The series stationarity was checked using the augmented Dickey–Fuller test [55] which found the p-value to be $1.09e-08$. This is lower than 0.05, the commonly used threshold, meaning that we can set the d parameter to 0.

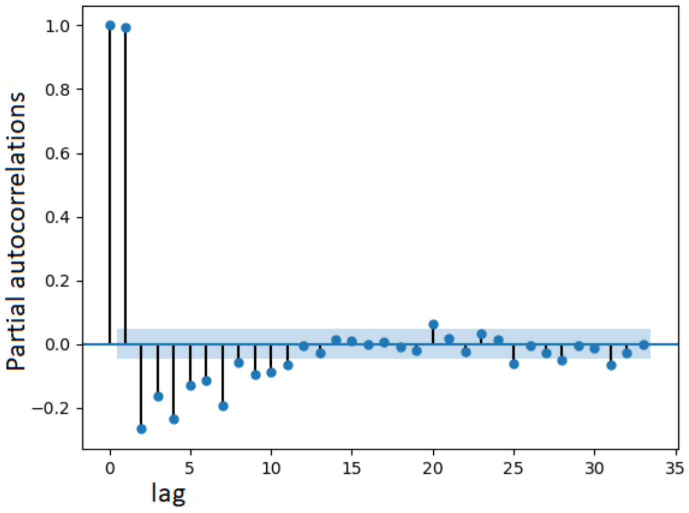

Figure 5. Partial autocorrelation plot for ARIMA

The partial autocorrelation plot (Figure 5) was analyzed to set the autoregression parameter (p). The significance region is confidently passed at 1, with a steep decline afterwards. The moving average parameter (q) is approximated from the autocorrelation plot (Figure 6) which suggests a value of around 20 would be a good start. After fitting ARIMA(1, 0, 20) the final 2 layers had p-value of 0.547 and 0.758, which meant that they were not significant enough, therefore we used 18 as the upper limit for q in our tuning.

Results. The results obtained for several values for $q : 5, 10, 15, 18$, are illustrated in Table 1, the best one being for ARIMA(1, 0, 15) with 1.42×10^7 MSE, 3056.7 MAE and 6.3% MAPE.

5.1.3. MLP

Settings. After some manual experiments we started with a MLP with 2 hidden layers $(150, 100)$ neurons, and a sliding window size of $n = 24$ (this is the window used to

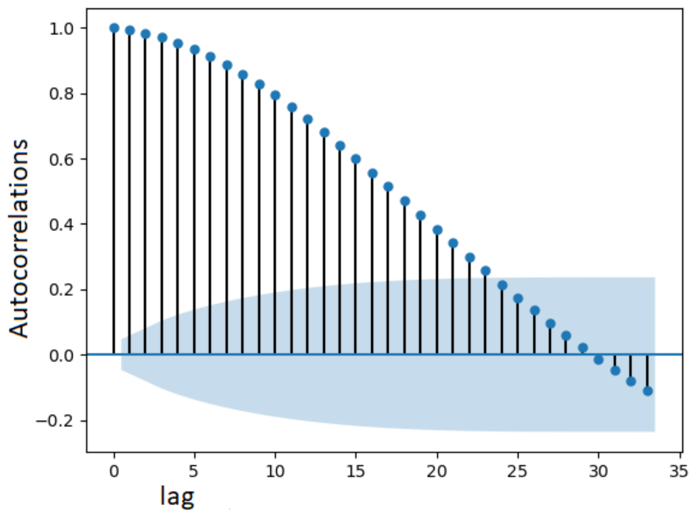

Figure 6. Autocorrelation plot for ARIMA

Table 1. ARIMA tuning – based on different values for *q* parameter

p	d	q	MSE / 10^6	prediction time
1	0	5	15.4	0.4 s
1	0	10	14.3	9.6 s
1	0	15	14.2	31 s
1	0	18	14.3	71 s

transform the time series data into a supervised dataset, meaning how many buckets are taken into account for each prediction, not the bucket width which was set at 10 min). To find an optimal combination of batch size and epoch numbers a 2d grid search was performed, and the results are presented in Figure 7. Batch size should ideally be a power of 2 for extra performance on GPU architectures, as some experiments were ran on Google Colab's cloud GPU[1]. Using lower batch size is more accurate but the training is slower [56]. As expected the lowest MSE is obtained for the lowest batch size (4), however it does not drop significantly at 8, regardless of epochs numbers. The selection of the epoch numbers is again a trade-off between the speed and the accuracy. We noticed that using a smaller number of epochs (50) the performance is not very good, while the difference between 100 and 250 is not very important, meaning that we can get a good approximation using a model with a epoch size of 100.

Additional experiments were done by adding Dropout layers on different values (0.2, 0.1, 0.05), however it did not improve performance. These are generally used to prevent over-fitting, when the network is too big, the data is scarce, or the training is done for too long [57], which was not the case for this experiment.

Various optimizers and activation functions were tested, and from these Adadelta optimizer and ReL (Rectified Linear) activation function were selected. ReL activation function is also the default recommendation [47] for modern neural networks, because it is non-linear while preserving many advantages of linear functions that make them generalize well. Although the ADAM optimizer is widely used in research, there is no consensus on which is the

[1]https://colab.research.google.com

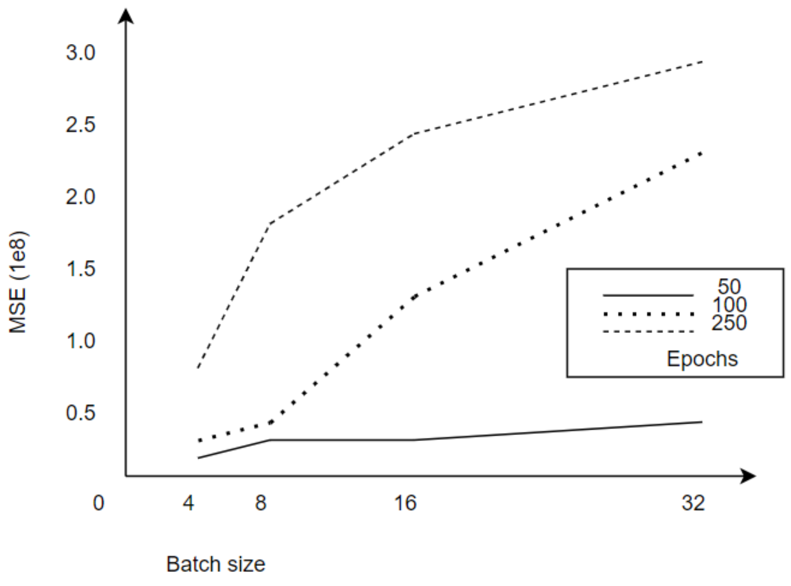

Figure 7. Grid search depending on the epoch number and the batch size

optimal one [47], therefore we choose Adadelta, which in our experiments performed better – 1.92×10^7 vs. 2.58×10^7 values for MSE.

A comprehensive grid search was performed for sliding window size and number and content of hidden layers, of around 90 combinations. Some of the best results are presented in Table 2.

Table 2. MLP tuning – based on different combinations of sliding window size and number and content of hidden layers

Sliding window	Hidden layers	MSE / 10^6
4	(100, 50, 25, 20, 10)	18.8
4	(100, 50, 50, 20, 10)	18.8
8	(100, 50, 25, 20, 10)	17.0
8	(150, 50, 50, 50, 50, 10)	17.2
8	(50, 50, 50, 50)	18.3
16	(10, 20, 30, 40, 50)	18.1
16	(100, 20, 20, 20, 10)	18.4

Results. The final parameters chosen for the proposed MLP model were: Adadelta optimizer with ReL activation function, a sliding window of size 8 with 5 hidden layers of size: $100, 50, 25, 20, 10$.

5.1.4. CNN

Settings. Firstly, a baseline model was selected through manual experimentation. This had the following structure: input of size 20, a 1d convolutional layer, a max pooling layer, a flatten layer, a dense layer of size 150 and the output layer. The same batch size, epoch number grid search was performed and it yielded similar results to those reported in Figure 7 for MLP. This was followed by iterating the same optimizers and activation function which resulted in our selection of Adadelta and softplus.

A grid search was again performed in order to find out the optimal sizes for sliding window, hidden layers and their neurons (see Table 3). This proves our assumption that CNNs can extract better features from larger sliding window as best results were obtained with a window of 128 as opposed to 8 for MLPs.

Table 3. CNN tuning – based on different combinations of sliding window size and number and content of hidden layers

Sliding window	Hidden layers	MSE / 10^6
8	(25, 10, 5)	35.3
64	(100, 20, 10, 5)	35.0
128	(100, 20, 10, 5)	21.0
128	(300, 50)	22.2
128	(10, 10, 10)	23.4
256	(100, 20, 10, 5)	23.7

Results. The parameters selected for the CNN model were: Adadelta optimizer, softplus activation function, a window of size 128 and 4 hidden layers of size: $100, 20, 10, 5$.

5.1.5. CNN-LSTM hybrid

Settings. The starting values for some parameters were influenced by the research done by Lin et al. [19]: a convolutional layer with 32 CNN filters, a max pooling layer, a LSTM layer with a couple of hundred units. In order to feed the output of the convolutional layers into the LSTM layer the input was broken into multiple sequences. This provided the time dimension which LSTM input shape specifies, as the sequences are arranged in a temporal order.

A similar search as for the previous model was performed and as a result we selected Adadelta optimizer and ReL activation function.

While searching for the size of the LSTM layer, we observed a trend where error value would become very large after a couple of epochs, of approximately $1.7 \cdot 10^{27}$. This might be linked with a gradient explosion [47], which causes a network to become unstable because of an increase in the number or values of the gradients with which the inputs are multiplied. Therefore, we applied a common solution, to rescale elements in a gradient vector if their norm exceeds 1, which has solved the issue.

A search was then performed for different combinations of sliding window size (which is transformed into a 2D structure, the input shape of the algorithm), CNN sequences and LSTM units, and the most important results are shown in the Table 4.

Table 4. CNN-LSTM tuning – based on different combinations of input shape and size of LSTM layer

Input shape	LSTM units	MSE / 10^6
(20, 15)	500	246.0
(16, 16)	150	98.3
(16, 16)	500	90.2
(12, 12)	750	93.6
(12, 12)	500	97.3
(8, 8)	500	370.1

It can be observed that the MSE values are quite larger than those reported in the validation of previous models. The reason for this is that the amount of data used for training becomes smaller as we increase the sliding window size. There was also a lot of variance for different runs of the same configuration.

Results. From the previously described experiments we selected the following parameters for this model: Adadelta optimizer, ReL function, $(16, 16)$ input shape (sliding window size equal to 256) with 500 LSTM units.

5.2. Evaluation

The evaluation was done on both Japanese and German Wikipedia traces with two time windows on each, 10min and 15min, thus obtaining 4 data sets. The sliding window size was slightly scaled when evaluating models on the 15min window with a 0.66 ratio to account for the different time ranges in the data set.
This evaluation process of retraining and testing the models has been repeated 10 times to account for the random weight initialization. The results obtained using deep learning models were averaged and then compared to baselines and to each other, and the results could be seen in Table 5.

Table 5. The MSE based comparison of the final results (the values are divided by10^6)

DataSet	Naive	ARIMA	MLP	CNN	CNN-LSTM
Jp10	20.2	15.2	8.5	11.7	7.2
Jp15	87.8	56.6	31.0	35.0	50.2
De10	16.7	10.3	5.1	10.5	21.3
De15	77.4	43.3	17.2	35.4	65.7

If we compare the results across across all datasets then we may conclude that MLP performed very well, consistently passing both baselines. On average, the MLP model was 49% more accurate than the classical ARIMA method which also indicates a better performance than CloudInsight [9] which obtained a 12% improvement over ARIMA on the same dataset.

CNN performed a bit worse, but still managed to beat the baselines in 3 out of 4 cases, while being very close on the other one.

CNN-LSTM has been very inconsistent. On Jp10 dataset it obtained the best result, beating MLP but this performance has not been repeated. In the De10 and De15 experiments it did not even beat ARIMA performance. This has not been improved even after multiple measurements or epochs, as seen on the loss plot from Figure 8, which indicates that the loss improves very little after 100 epochs.

Computational overhead. An aspect which should be noted is the computational overhead of the proposed models. The prediction time for the ARIMA model varies from 0.4 s to 31 s for the most accurate one (see Table 1). The time in which the deep learning models make a prediction (once trained) is much shorter: 0.16 s for MLP, 0.21 s for CNN and 0.24 s for CNN-LSTM.

Best performer: MLP. The comparison revealed this model to be the best performer, beating both ARIMA baseline and the CloudInsight hybrid model. A more detailed comparison with the classical method can be seen in Table 6 taking into account all error metrics. A plot of the predicted traffic on the Jp10 dataset can be seen in Figure 9.

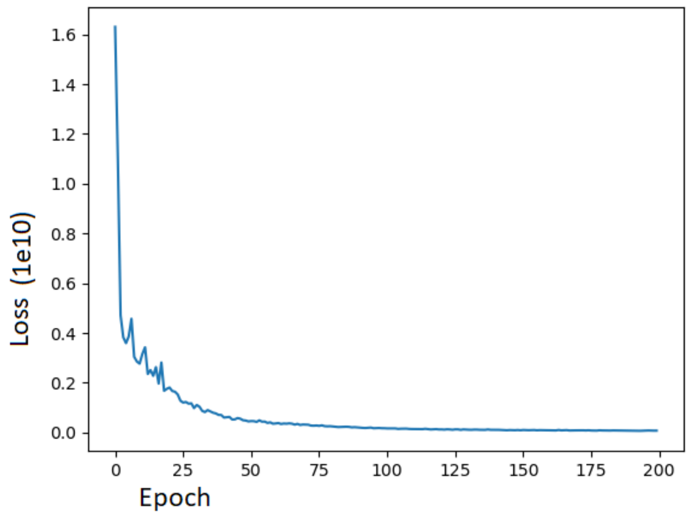

Figure 8. Training loss for CNN-LSTM on De15 with different numbers of epochs

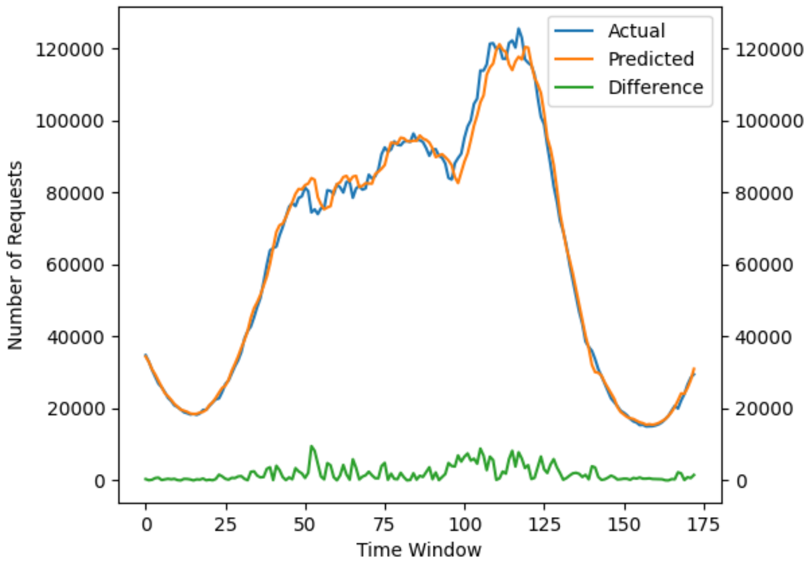

Figure 9. Actual vs. Predicted traffic for about a day on the Jp10 dataset using MLP

Table 6. MLP vs ARIMA, compared based on MSE/MAE/MAPE

	ARIMA			MLP		
DataSet	MSE	MAE	MAPE	MSE	MAE	MAPE
Jp10	15.2E6	3056	6.3%	8.5E6	1960	2.9%
Jp15	56.6E6	6124	8.8%	31.0E6	3540	3.4%
De10	10.3E6	2517	7.6%	5.1E6	1583	3.4%
De15	43.3E6	5606	13.4%	17.2E6	2787	3.9%

5.3. Threats to validity

As with any experiment based analysis, the reported results and conclusions could be subject to certain threats to validity [58]. The followings are the major threats to the validity of our work and the ways we tried to mitigate them.

- Construct validity – For our purposes we assume that the number of requests in an interval is a satisfactory prediction metric. This tends to be enough because a microservice should fit a specific business case therefore having homogeneous requests. The results might not be as accurate when a single microservice type handles widely different requests.
- External validity – We propose a prediction model building methodology which can be customized to the specific microservice deployment it is used on, because the Δt window should be chosen after performance benchmarking the selected application (see Section 3).
- Internal validity – The metric we chose (number of requests) is independent of the prediction result. The other options (e.g., CPU utilization, average response time) would change depending of the scaling actions performed and then influence further decisions, causing a small error to propagate in time.
  We used scaling on input data to improve accuracy and training time. In a real world scenario scaling could still be done using historical bounds which would be updated periodically. Values which are out of these bounds could impact accuracy [59]. In the case of a sudden burst of requests with no historic precedent the measurement of the metric may be impacted. For example if there is a bottleneck of requests waiting to be processed they may not even be counted. However, in a practical deployment the system would eventually scale to handle the requests but it would take multiple scale out commands instead of just one (which is the norm in non exceptional scenarios).
- Reliability – The selected dataset is publicly available [17] and has been used by other researchers for the same goal [9].
  The selected models have been implemented and evaluated using Keras, scikit-learn, and statsmodels libraries. The measurements analysis was in general based on their default settings and on repeating the processes, but an extended analysis of the possible measurements errors could reveal the need of some additional adaptations. From our observations, the MSE value ranges do not wildly fluctuate on multiple measurements, which is also indicated by calculating confidence intervals. For example, on a random re-evaluation with a larger number of repetitions (30) of MLP on De10 dataset with a 95% confidence level the resulting confidence interval was $6.6 \pm 0.67(1e6)$, which still convincingly outperforms the baselines.
  During experiments we have chosen k-fold cross validation with $k = 3$, but we are aware of the fact that a higher value for k could estimate a more accurate confidence interval [60–62]. Our choice was justified by the impact on the computational time of a higher value for k. We tried various settings and layer distributions for some of the more complex models (CNN and CNN-LSTM) and choosing for example $k = 10$ would have led to a much higher asymptotical computational complexity. Still, we acknowledge that would be worth investigating the results that could be obtained using a much higher value of k for cross validation.

5.4. Results analysis

In order to find an efficient model for a proactive auto-scaler for microservices, we have started by analysing the most suitable steps, and we arrived to a methodology adapted to the microservices specifity.

Inside the proposed methodology we compared a naive and a classical time series method – ARIMA, with three deep learning models, MLP, CNN, CNN-LSTM, over two traces of Wikipedia traffic data and two time windows (of 10 and 15 minutes).

This analysis emphasized that MLP(Multilayer Perceptron) shows considerable improvements in performance over the classical method, of around 49% in MSE, which is also better than some state of the art models currently used for this task, like the council of experts employed by CloudInsight [9].

It also showed that the sophisticated hybrid CNN-LSTM can obtain great results (having the best performance on Jp10), however it requires considerably more tuning and training time. Given a larger data trace and tuning effort, it might become the most accurate model.

MLPs are much faster to train than the other deep learning models which facilitates the periodic workload data update a practical application might need.

All these recommend MLP as the best choice for application in a practical proactive auto-scaler. This model was selected as the most appropriate from our implementation (based on the selected collection of models) of the methodology described in Section 3. Still our investigation maybe also seen as a starting point for other applications of this methodology.

5.5. Practical usage

In order to emphasise a possible practical usage of the model presented in this research, we developed a proof of concept implementation of a predictive scaling tool, which is available at https://github.com/StefanSebastian/MicroserviceMonitoring/tree/master/monitor_scaler_app.

The tool, modelled in Figure 10, was designed to simplify the process of monitoring and automatic scaling as much as possible. The main components are:

– a server application which consumes the data stream from all microservice instances (through Kafka message queue) generates various statistics and stores aggregates into the local database for training the prediction model,
– a dashboard monitor application which displays all active microservices and their performance, and

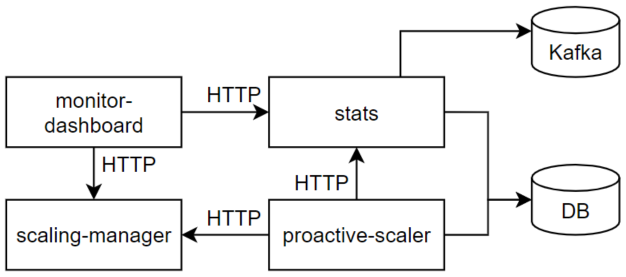

Figure 10. The schema of the proof of concept scaling tool

– a scaling manager which is responsible with starting or stopping instances of the service (in the demo implementation it executes Docker commands to start or stop containerized microservices)
– a proactive scaler which uses historic data to predict traffic patterns (containing a Python implementation of the proposed MLP model) and generates scaling decisions, which are then forwarded to the scaling manager.

An additional Java client is also provided which captures traffic from Spring Cloud microservice implementations and puts onto a kafka queue which feeds data into the scaling system.

This scaling tool was tested on a simplistic microservice system (https://github.com/StefanSebastian/MicroserviceMonitoring/tree/master/demoapp) built on the Spring Cloud stack: a Zuul load balancer, an Eureka name server, and a microservice which simulates workloads, and was shown to work on a manually prepared scenario. The scenario consisted of one spike of traffic repeated over and over again, for which we compared the system performance of reactive and predictive scaling approaches. The conclusion was that in the proactive approach the average processing time of the system was 14% better.

6. Conclusions

The paper proposes a methodology for microservice oriented workload prediction and analyzes whether deep learning models are appropriate to be used as a prediction model for this kind of data. The methodology is adapted to practical microservice demands, such as the metric selection of the number of requests, which are not influenced by the scaling prediction, and the prediction in time intervals of an order of minutes, with a buffer window in which the services can be deployed.

An empirical investigation was conducting in order to find the most appropriate deep learning model to be used for a microservice proactive auto-scaler. The tests and the comparison analysis led to the conclusion that considering the accuracy, but also the computational overhead and the time duration for prediction, MLP (MultiLayer Perceptron) model qualifies as a reliable foundation for the development of proactive micro-service scaler applications.

Future plans include investigation of other models, but also development of a more complex proof of concept project that considers realistic scenarios, with varied traffic patterns over a longer period of time to showcase the accuracy of the proposed tool.

References

[1] T. Erl, *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, 2nd ed. Springer International Publishing, 2016.

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly Media, 2021.

[3] *2020 Digital Innovation Benchmark*, Kong Inc, 2019. [Online]. https://konghq.com/resources/digital-innovation-benchmark-2020/ Released on konghq website.

[4] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca et al., "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Proceedings of 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 179–182.

[5] R.V. Rajesh, *Spring Microservices.* Packt Publishing, 2016.

[6] R.N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using ARIMA model and its impact on cloud applications' QoS," *IEEE Transactions on Cloud Computing*, Vol. 3, No. 4, 2015, pp. 449–458.

[7] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi et al., "Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers," in *Proceedings of 2010 IEEE International Conference on Services Computing*, 2010, pp. 514–521.

[8] M.S. Aslanpour, M. Ghobaei-Arani, and A. Toosi, "Auto-scaling web applications in clouds: A cost-aware approach," *Journal of Network and Computer Applications*, Vol. 95, 07 2017, pp. 26–41.

[9] I.K. Kim, W. Wang, Y. Qi, and M. Humphrey, "Cloudinsight: Utilizing a council of experts to predict future cloud application workloads," in *Proceedings of the 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 41–48.

[10] J. Kumar and A.K. Singh, "Workload prediction in cloud using artificial neural network and adaptive differential evolution," *Future Generation Computer Systems*, Vol. 81, 2018, pp. 41–52.

[11] J.C.B. Gamboa, "Deep learning for time-series analysis," *CoRR*, Vol. abs/1701.01887, 2017. [Online]. http://arxiv.org/abs/1701.01887

[12] P. Udom and N. Phumchusri, "A comparison study between time series model and ARIMA model for sales forecasting of distributor in plastic industry," *IOSR Journal of Engineering (IOSRJEN)*, Vol. 4, No. 2, 2014, pp. 32–38.

[13] K.I. Stergiou, "Short-term fisheries forecasting: comparison of smoothing, ARIMA and regression techniques," *Journal of Applied Ichthyology*, Vol. 7, No. 4, 1991, pp. 193–204. [Online]. https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1439-0426.1991.tb00597.x

[14] J. Zhu, R. Zhang, B. Fu, and R. Jin, "Comparison of ARIMA model and exponential smoothing model on 2014 air quality index in Yanqing County, Beijing, China," *Applied and Computational Mathematics*, Vol. 4, No. 6, 2015, pp. 456–461.

[15] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *Proceedings of 2012 IEEE Network Operations and Management Symposium*, 2012, pp. 1287–1294.

[16] M.D. Syer, W. Shang1, Z.M. Jiang, and A.E. Hassan, "Continuous validation of performance test workloads." *Automated Software Engineering*, Vol. 24, 3 2016, pp. 189–231.

[17] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, Vol. 53, No. 11, July 2009, pp. 1830–1845.

[18] J. Brownlee, *Deep Learning for Time Series Forecasting: Predict the Future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, 8 2018.

[19] T. Lin, T. Guo, and K. Aberer, "Hybrid neural networks for learning the trend in time series," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 2273–2279.

[20] X. Tang, "Large-scale computing systems workload prediction using parallel improved LSTM neural network," *IEEE Access*, Vol. 7, 2019, pp. 40 525–40 533.

[21] Y. Zhu, W. Zhang, Y. Chen, and H. Gao, "A novel approach to workload prediction using attention-based LSTM encoder-decoder network in cloud environment," *EURASIP Journal on Wireless Communications and Networking*, 2019, pp. 1–18.

[22] Q. Zhang, L.T. Yang, Z. Yan, Z. Chen, and P. Li, "An efficient deep learning model to predict cloud workload for industry informatics," *IEEE Transactions on Industrial Informatics*, Vol. 14, No. 7, 2018, pp. 3170–3178.

[23] *PlanetLab – An open platform for developing, deploying, and accessing planetary-scale services.* [Online]. https://planetlab.cs.princeton.edu Read October-2020.

[24] D. Jacobson, D. Yuan, and N. Joshi, *Scryer: Netflix's Predictive Auto Scaling Engine*, 2013. [Online]. https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270 Read 17-October-2020.

[25] *Why You Can't Talk About Microservices Without Mentioning Netflix*, SmartBear Software, (2015, December). [Online]. https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/ Read October-2020.

[26] J. Bar, *New-Predictive Scaling for EC2, Powered by Machine Learning*, (2018, November). [Online]. https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/ Read October-2020.

[27] *Autoscaling guidance – Best practices for cloud applications*, Microsoft, (2017, May). [Online]. https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling Read 17-October-2020.

[28] *Autoscaling groups of instances*, Google, 2014. [Online]. https://cloud.google.com/compute/docs/autoscaler Read October-2020.

[29] P. Jamshidi, C. Pahl, N.C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, Vol. 35, No. 3, 2018, pp. 24–35.

[30] *Spring Cloud Netflix*. [Online]. https://cloud.spring.io/spring-cloud-netflix/reference/html/ Read October-2020.

[31] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, "Research on auto-scaling of web applications in cloud: Survey, trends and future directions," *Scalable Computing: Practice and Experience*, Vol. 20, 05 2019, pp. 399–432.

[32] A.A. Bankole and S.A. Ajila, "Predicting cloud resource provisioning using machine learning techniques," in *Proceedings of the 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2013, pp. 1–4.

[33] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–32.

[34] R. Shumway and D. Stoffer, *Time Series Analysis and Its Applications with R Examples*, 3rd ed. Springer, 2011.

[35] M. Alloghani, D. Al-Jumeily, J. Mustafina, A. Hussain, and A.J. Aljaaf, *A Systematic Review on Supervised and Unsupervised Machine Learning Algorithms for Data Science*. Cham: Springer International Publishing, 2020, pp. 3–21. [Online]. https://doi.org/10.1007/978-3-030-22475-2_1

[36] T.G. Dietterich, "Machine learning for sequential data: A review," in *Structural, Syntactic, and Statistical Pattern Recognition*, T. Caelli, A. Amin, R.P.W. Duin, D. de Ridder, and M. Kamel, Eds. Berlin, Heidelberg: Springer, 2002, pp. 15–30.

[37] G. Bontempi, S. Ben Taieb, and Y.A. Le Borgne, *Machine Learning Strategies for Time Series Forecasting*. Springer Berlin Heidelberg, 01 2013, Vol. 138, pp. 62–67.

[38] M. Smeets, *Microservice framework startup time on different JVMs*, 2019. [Online]. https://technology.amis.nl/languages/java-languages/microservice-framework-startup-time-on-different-jvms-aot-and-jit/ Read 26-June-2021.

[39] R. Frank, N. Davey, and S. Hunt, "Time series prediction and neural networks," *Journal of Intelligent and Robotic Systems*, 2001, pp. 91–103.

[40] A. Botchkarev, "Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology," *Interdisciplinary Journal of Information, Knowledge, and Management*, Vol. 14, 2019, p. 045–076.

[41] C.D. Lewis, *Industrial and business forecasting methods: A practical guide to exponential smoothing and curve fitting*. London(U.A.): Butterworth Scientific, 1982.

[42] S.L. Ho and M. Xie, "The use of ARIMA models for reliability forecasting and analysis," *Computers and Industrial Engineering*, Vol. 35, No. 1–2, Oct. 1998, p. 213–216.

[43] A.O. Adewumi and C.K. Ayo, "Comparison of ARIMA and Artificial Neural Networks models for stock price prediction," *Journal of Applied Mathematics*, Vol. 2014, 03 2014, pp. 1–7.

[44] S. Siami-Namini, N. Tavakoli, and A. Siami Namin, "A comparison of ARIMA and LSTM in forecasting time series," in *Proceedings of 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 1394–1401.

[45] V.R. Prybutok, J. Yi, and D. Mitchell, "Comparison of neural network models with ARIMA and regression models for prediction of Houston's daily maximum ozone concentrations," *European Journal of Operational Research*, Vol. 122, No. 1, 2000, pp. 31–40. [Online]. https://www.sciencedirect.com/science/article/pii/S0377221799000697

[46] T.J. Sejnowski, *The Deep Learning Revolution.* The MIT Press, 2018.

[47] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, Adaptive Computation and Machine Learning series. MIT Press, 2016. [Online]. https://books.google.ro/books?id=Np9SDQAAQBAJ

[48] R.D. Reed and R.J. Marks, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks.* Cambridge, MA, USA: MIT Press, 1998.

[49] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, Vol. 9, No. 8, Nov. 1997, p. 1735–1780.

[50] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[51] F. Chollet and et al., *Keras*, GitHub, 2015. [Online]. https://github.com/fchollet/keras

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, Vol. 12, 2011, pp. 2825–2830.

[53] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with Python," in *Proceedings of the 9th Python in Science Conference*, 2010.

[54] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Machine Learning Research, Y.W. Teh and M. Titterington, Eds., Vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. https://proceedings.mlr.press/v9/glorot10a.html

[55] Y.W. Cheung and K.S. Lai, "Lag order and critical values of the augmented Dickey–Fuller test," *Journal of Business & Economic Statistics*, Vol. 13, No. 3, 1995, pp. 277–280.

[56] N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," in *Proceedings of 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26*, 2017.

[57] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, Vol. 15, No. 56, 2014, pp. 1929–1958.

[58] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, Vol. 14, 2008, pp. 131–164.

[59] S. Nayak, B.B. Misra, and H.S. Behera, "Impact of data normalization on stock index forecasting," *International Journal of Computer Information Systems and Industrial Management Applications*, Vol. 6, No. 2014, 2014, pp. 257–269.

[60] J.D. Rodriguez, A. Perez, and J.A. Lozano, "Sensitivity analysis of k-fold cross validation in prediction error estimation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 32, No. 3, 2010, pp. 569–575.

[61] R.R. Bouckaert, "Estimating replicability of classifier learning experiments," in *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04. New York, NY, USA: Association for Computing Machinery, 2004. [Online]. https://doi.org/10.1145/1015330.1015338

[62] M. Huk, K. Shin, T. Kuboyama, and T. Hashimoto, "Random number generators in training of contextual neural networks," in *Proceedings of 13th Asian Conference on Intelligent Information and Database Systems*, N.T. Nguyen, S. Chittayasothorn, D. Niyato, and B. Trawiński, Eds. Cham: Springer International Publishing, 2021, pp. 717–730.