Using Machine Learning to Predict the Exact Resource Usage of Microservice Chains

Javid Taheri  
Karlstad University, Sweden  
Queen’s University Belfast, UK  
Sweden and UK  
javid.taheri@kau.se  
j.taheri@qub.ac.uk

Arvid Gördén  
Karlstad University  
Sweden  
arvid@bag.org

Auday Al-Dulaimy  
Mälardalen University  
Dalarna University  
Sweden  
auday.aldulaimy@mdu.se

ABSTRACT

Cloud computing offers a wide range of services, but it comes with some challenges. One of these challenges is to predict the resource utilization of the nodes that run applications and services. This is especially relevant for container-based platforms such as Kubernetes. Predicting the resource utilization of a Kubernetes cluster can help optimize the performance, reliability, and cost-effectiveness of the platform. This paper focuses on how well different resources in a cluster can be predicted using machine learning techniques. The approach consists of three main steps: data collection and extraction, data pre-processing and analysis, and resource prediction. The data collection step involves stressing the system with a load-generator (called Locust) and collecting data from Locust and Kubernetes with the use of Prometheus. The data pre-processing and extraction step involves extracting relevant data and transforming it into a suitable format for the machine learning models. The final step involves applying different machine learning models to the data and evaluating their accuracy. The results illustrate that different machine learning techniques can predict resources accurately.

CCS CONCEPTS
• Computer Science → Cloud Computing.

KEYWORDS
Cloud computing, Resource management, Auto-scaling, Kubernetes, Microservice, Machine learning

ACM Reference Format:

1 INTRODUCTION

Microservices is a software architecture that has risen in popularity over the years. It is a design pattern that splits a complex system into smaller, independent, and loosely coupled services. Each service handles a specific functionality or domain and interacts with other services through well-defined interfaces. In this way, microservices enable faster and more reliable delivery of applications, as each service can be developed, tested, deployed, and scaled independently. Microservices also improve fault tolerance and resilience of the system, as failures in one service do not affect the whole system. Containerization is a technology that facilitates the implementation of microservices, as it allows for creating isolated and lightweight environments for running each service [4].

Containerization is a way of virtualizing applications so that they run in their own environments without interfering with each other, but still use the same operating system resources. A container is a package that has everything an application needs to run, including libraries, data, configuration files, etc. [6] [11]. Containerization differs from full virtualization by focusing on the operating system layer instead of the hardware layer. This means that containers are more lightweight and efficient than virtual machines. Some examples of technologies that enable containerization and container management are: Docker, Kubernetes, and Red Hat OpenShift [11].

Kubernetes allows for autoscaling through its Horizontal Pod Autoscaler. Scaling in a cloud system can be primarily performed in two ways: Horizontal- and Vertical-Scaling. Horizontal scaling involves adding or removing entire containers or virtual machines. The advantage of this method is that it can handle large variations in resource demand. The disadvantages are that it is slower because it takes time to deploy a new virtual machine and that the time delay is not consistent. Vertical Scaling involves changing the size of existing resources, such as the CPU or memory, allocated to an already deployed virtual machine. The drawback of this method is that it has a limited range of possible control actions [5]. Almost all distributed systems allow horizontal scaling, whereas, vertical scaling is only supported by a few platforms (e.g., VMware vSphere) and to a limited extent in most cases. One of the challenges of developing applications using a microservice architecture is to predict the number of exact resources (CPU, Memory, etc.) each service needs. If a service needs more resources, it may become faster, more responsive, and more successful. On the other hand, if a service has too many resources, it may waste money and energy. To address this challenge, some microservice architectures use autoscaling techniques to adjust the resources allocated to each service based on its current or historical demands. Autoscaling can be reactive or proactive. Reactive autoscaling monitors the performance metrics of each service and scales up/down when predefined thresholds are met. For example, if the CPU utilization of a service exceeds 80%, more instances of that service are created to handle the load. Reactive autoscaling is simple, but may introduce latency or downtime.
during scaling operations. Proactive autscaling tries to predict the future demand of each service and scales up/down in advance [1] [2]. For example, if a service expects a surge of traffic during a particular time of the day, more instances of that service are created beforehand to avoid congestion. Proactive autscaling is more complex and can affectively reduce latency or downtime during the scaling operations, but requires sophisticated procedures to accurately predict future resource demands. [7] [10].

The contributions of this paper lay in answering the following research questions:

- How well can machine learning algorithms be used to accurately predict resource utilization in a microservice application?
- What fraction (how much) of data is enough for training to accurately predict future resources?

The rest of this paper is organized as follows. Section 2 highlights the related work. Section 3 lays out our proposed procedure. Section 4 reflects our results and analysis after evaluating our approach, followed by Section 5 that concludes this paper.

2 RELATED WORK

Previous research on using AI based models for autscaling is summarized and described in this section.

In [8], a proactive framework based on machine learning is presented. The proposed solution is an autcaler that scales horizontally. Four different machine learning models were compared: ARIMA, LSTM, Bi-LSTM and transformer based models. The model that performed the best out of all of them was the Informer model, which is a transformer based model.

The work in [3] presents a system that centralizes cluster autscaling and resource management. It offers a low-latency automated system for managing containers and assessing resiliency for dynamic systems. The system predicts the load using a Bi-LSTM and periodically updates the autscaling policy for cluster performance. The system is proactive and performs both horizontal and vertical scaling. The study also compared three different algorithms: ARIMA, Holt Winters and Bi-LSTM where Bi-LSTM performed the best.

The authors in [9] proposed a proactive horizontal scaling solution that makes use of different machine learning methods to optimize resources. This is done by calculating the accuracy of each model and choosing the one with the highest accuracy. Three models were used, namely: HTM, LSTM and AR. AR was used the most and LSTM short thereafter.

Many of these studies propose similar solutions and are usually only for horizontal scaling. Many machine learning methods are also the same, leaving a gap for examining other models. Usually, only one resource metric is also used for predictions, which is CPU usage. However, in the works discussed above, the system that is being tested is primarily small and does not contain several deployments. To address these research gaps, we proposed an approach that works on industry approve complex microservice such as Google Online Boutique with 11 independent microservices; examined more machine learning approaches; and predicted more resources (besides CPU that was a common factor in all previous works).

3 THE PROPOSED APPROACH

This section describes methodology parts of the proposed approach, that is, data collection, pre-processing, and machine learning.

3.1 Data collection

The collection of the data can be separated into different parts. Firstly, before swarming the cluster, the Kubernetes API needed to be invoked to make sure that everything is running correctly and/or to make changes to the cluster. A python file was created for handling the Kubernetes API communication. Within this file, two classes were created. One called “Patcher” whose main objective and functionality was to perform the so-called patch operations on Kubernetes objects, for example to set or adjust the amount of replicas or resources like CPU or Memory limits. The second class was called “Reader” whose functionality was to perform “get” operations, for example getting the amount of replicas for one pod or to watch the cluster for changes and tell the program to halt until a change was made/registered. Both of these classes needed to load in the kube_config file to be functional.

Secondly, the Locust swarm needed to be started. The Locust web UI has an internal API that was visible in the network tab when using developer tools in a browser. This was utilized when making the management of locust swarms available programmatically. In order for this, the python requests library was used which simplifies the usage of HTTP POST and HTTP GET. The class that handled Locust swarms was called “LocustManager” and it contained three methods, including its constructor. These were: start_swarm to compose a dictionary containing the user count, spawn rate, host and total run time for the swarm workers; download_report to download the results from the locust swarm master and saved it as an HTML file. These classes together with the Patcher and Reader class were then used in a python file whose purpose was to start new tests and download the data. Since many tests would be run after each other, blueprints for different collections of tests were made. These were defined in a YAML-file where the user can define specifications for patching the Kubernetes cluster and define the specifics for the locust swarm (such as run time, user count and spawn rate). A sample blueprint is as following:

```yaml
test_cases:
  - k8: ()
    locust:
      host: http://172.16.19.100:32000
      run_time: 241
      spawn_rate: 100
      user_count: 100
      test_number: 0
```

A python module for quickly creating blueprints was also created to iterate over a number of defined user counts and create different test cases. The run time was calculated by using a formula \((\text{round} (\text{users}/\text{locust}\_\text{spawn}\_\text{rate}) + 180)\) that we empirically improvised. It predicts the amount of time requires to reach stable outputs/behaviour from the Kubernetes cluster, as well as the application (Google Online Boutique).

When a swarming round was done (i.e., running it and downloading the HTML result-file), we converted the result into a JSON
format to make the extraction and parsing of the data easier (using etree from the lxml library). After each swarm, data from Prometheus needed to be collected as well in order to obtain information on CPU usage, memory usage, network usage, etc. The Python library for Prometheus to collect the data (using PromQL queries) fetched the data as a dictionary and then converted it to another JSON-file for processing.

### 3.2 Data pre-processing

When the gathering of the data was done, the correct data had to be chosen and extracted and put into a form that would be of benefit for the machine learning models. This was done by loading the JSON-files for each swarm into a Pandas dataframe and placing the desired points of data into a Pandas dataframe together with the name of the pod and the user-count for that particular swarm. The data points that were extracted:

- CPU usage
- Memory
- Memory Irate
- Network receive
- Network transmit
- Replicas

First, the data from the Locust and Prometheus JSON-file was retrieved. This was done by loading the JSON into a python dictionary and then searching through the dictionary for each desired metric and creating a new dictionary with all the datapoints for each metric. The new dictionaries were then converted into Pandas dataframes by utilizing the from_dict method that is present in the Pandas library. After this, all dataframes with the different metrics were merged into one and the user-count from the Locust dictionary was inserted into the dataframe as well. When this was done a second dataframe was made that would contain the normed data points. The normalization method that was used was min-max normalization which uses the formula: $x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$, where $x'$ is the normed value, $x$ is the actual value, $x_{\text{min}}$ is the minimum value of the data and $x_{\text{max}}$ is the maximum value of the data.

This method of normalization turns the data with the minimum value into 0 and the maximum value into 1 and the rest in between. These were then saved into a CSV-file for quick and easy access at any time.

Before applying machine learning to the data, the pod names needed to be encoded to a non-string value since the machine learning methods used from scikit-learn cannot use string values. This was achieved by using the class LabelEncoder which is build in the scikit-learn to transform a string into an integer which it can later decode as well.

### 3.3 Machine Learning

Three different machine learning methods were used and compared to predict resources. These were:

- Multilayer perceptron
- Support Vector Machine with linear kernel
- Support Vector Machine with polynomial kernel

The rationale behind choosing these choices is because they address the gaps in the related studies when using machine learning methods. When using machine learning in this project, the input variable chosen was amount of users and the features were: CPU, Memory, Memory Irate, Network transmit and Network receive. The reason these were chosen is because they are relevant resources that could be bottlenecks for different pods. When training was being done, a unique model was trained for each combination of feature (pod name, user step and machine learning algorithm). It means the input would always be user count and the output would be one of the features mentioned earlier. This was done to achieve a higher accuracy for the models as this have previously been tested by using all features as output. User-step, in this context, reflect how number of users were incremented between test scenarios; for example, User-step = 100 implied datapoints were collected for users = 200, 300, 400, etc. This was done to evaluate how much of the training data would be needed to accurately train machine learning models, yet still achieve good accuracy on the predictions. The different user-steps that were used for training was: 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, and lastly 3000. To make this clearer, a unique model could be:

```python
feature: CPU
pod name: frontend
user step: 300
ML algorithm: SVM polynomial
```

while another could be:

```python
feature: Memory
pod name: currency-service
user step: 200
ML algorithm: SVM linear
```

The training was done on the normalized data and the predicted data was then un-normalized by using the reverse formula ($x = x' (x_{\text{max}} - x_{\text{min}}) + x_{\text{min}}$).

Hyperparameter tuning was performed on each model to find the best training parameters for each particular data set. The parameter grid used for multilayer perceptron was:

```python
{'hidden_layer_sizes': [(50,), (100,), (150,)],
 'activation': ['relu', 'tanh', 'logistic'],
 'alpha': [0.0001, 0.001, 0.05],
}
```

and for both kernels for support vector machine it was:

```python
{'C': [0.01, 0.1, 0.5, 1.],
 "degree": [3, 4, 5]
}
```

The reason for choosing these hyperparameters for optimization is that these are the ones that would have the most frequent changes when training different models. C and alpha are regularization parameters and help with over- and under-fitting. The motive behind choosing a small search space for the different machine learning algorithms was because of time constraint and resource constraint on my personal laptop; hyperparameter tuning could have become a bottleneck during training if set inappropriately. The method chosen for hyperparameter tuning was GridSearchCV which chooses a set of the cartesian product of the hyperparameters.
and trains, validates and calculates the accuracy of the chosen set until it finds the most accurate set of parameters.

4 PERFORMANCE EVALUATION

4.1 Experiment setup

4.1.1 Kubernetes cluster topology. The Kubernetes cluster used during this project was hosted on vSphere that was running on Ubuntu-based VMs, consisting of 1 master- and 4 worker-nodes. The specifications for the nodes are presented in Table 1. The VMs running Kubernetes was available on Karlstad University and could be accessed through VPN and SSH.

4.1.2 Demo application. The demo application that was used for testing and measuring is called Google Online Boutique. It is an online shop with 11 microservice, each performing a function in the whole process; for example, frontend provides a GUI (web-site) to browse and select items to purchase. The services are written in different programming languages like Python, Go, Java, and Node.js. The primary objective of the demo application is to showcase how to deploy, test, and monitor a microservice application with Kubernetes.

4.1.3 Programming environment. Python was the programming language that was chosen for this work due to its simplicity and richness in providing many machine learning libraries. The libraries we used during the work were:
requests for sending HTTP requests and handling their responses
• numpy for numerical computing and manipulating arrays
• pandas for data analysis and manipulation
• matplotlib for creating plots and visualizations
• kubernetes API for interacting with the Kubernetes cluster management system
• scikit-learn for machine learning and data mining

Jupyter notebook was used to for calculations and illustrating the graphs and tables from packages like pandas and matplotlib.

4.1.4 Presenting and visualizing the data. Matplotlib was used for visualizing and plotting the data. Figures for every resource were created that contained an axis for each pod (microservice). Each axis contained different graphs for every trained model and also

the 20 values that had been randomly sampled from the original data set.

4.2 Machine Learning approaches and results

Figures 1, 2, 3, and 4 present the predictions from all the different models as graphs and the 20 random original points as blue dots. In these figures, the x-axis represents the user count, and the y-axis the predicted/actual value for each resource. As it can be seen, each model (e.g., SVM) is trained using various user-steps; this is to investigate how much data (an in what resolution) is actually required to properly train a machine learning SVM model.

When measuring the prediction error, 20 random samples from the whole dataset were used. The error was simply calculated by subtracting the predicted value from the actual value. From these error values, two measures of error were obtained: mean squared
5 DISCUSSION AND ANALYSIS

The results in previous section, we showed that SVM linear usually performs quite severely and the model generally underfits the training data. This is made clear in Figure where SVM linear is far off for several of the original data points. There are, however, figures where it performs better, for example 2 for the frontend and productcatalogservice pods. The case of the model performing badly is also supported by other figures showing that SVM linear has a generally high mean squared error and 95th percentile error. This is reinforced in Figure 9 where, for most pods, SVM linear’s MSE and 95th percentile error is higher than the other models.

MLP performed relatively well in regard to mean squared error and 95th percentile error. However, the figures in the previous section suggest that the model generally overfits the data. This is also made clear in Figure 1 but even more in Figure 6 where MLP takes statistical outliers and noise into consideration for payment service, ad service and shipping service pods. This would explain why it also sometimes outperforms SVM poly in some cases.

SVM poly performs the best overall when considering error measurements. All figures indicated that SVM poly has achieved a balance between underfitting and overfitting, especially as compared with SVM linear and MLP. This is despite the fact that in a few cases in Figure 1, it is shown that some models for SVM poly do not fit the original datapoints particularly well.

Combining the results from all figures to holistically evaluate these models, it can be observed that lower user-steps has a smaller error, and going above 1000 user-steps yields higher errors.
The results from the machine learning models suggest that when predicting resource utilization for Kubernetes, using SVM with a polynomial kernel usually results in acceptable predictions, that is, lower error while at the same time finding a balance when fitting the model. MLP could also be useful, but it usually overfits the data, and thus may produce misleading predictions. SVM linear should not be used as the model because it usually underfits the training data and leads to high error rates.

When looking at what user-step performed best, the discrepancy in lower x-axis in the figures should be addressed. These high errors are most likely a result of SVM linear underfitting, because it usually begins very far down away from the actual values of the data. One can also see a jump in errors at 6000 users. This could result from higher user steps like 1000, 2000, and 3000 having higher errors. From this, it seems that the lower user steps are better for training; however, there is no significant change in mean squared error between 200 and 1000 user steps. This means that any user step could probably be used between these values and still produce good predictions. Nevertheless, taking more datapoints into consideration is probably better, which is also quite intuitive.

5.1 Limitations
The largest limitations on the methodology used was the hyperparameter tuning as well as training a unique model for every combination as described earlier, since this resulted in training all the models and tuning their hyperparameters took a long time. The choice of hyperparameters could also be a limitation as choosing more hyperparameters might have produced better results, but at the same time, this would have increased time to train even more.
6 CONCLUSION AND FUTURE DIRECTIVES
This paper aimed to evaluate different machine learning methods used on data for resource utilization in a Kubernetes cluster when putting the cluster under different loads. This project was done by doing a literature review, setting up the testing environment in Kubernetes, running experiments, collecting data, extracting and analysing data, and learning about different techniques in machine learning. Data was collected by running different tests with the goal of analysing the data from these tests and using them with different machine learning models to predict resource utilization, followed by comparing them with each other. The test results and the comparison suggest that the best machine learning model (out of the three) was Support Vector Machine with a polynomial kernel, because it could well balance between over- and under-fitting the data, and also, performed generally well to predict all resource types.

As a future work, there are many alternatives that could be explored. One such direction would be to conduct more experiments with a larger number of users to evaluate the usability and effectiveness of the machine learning methods. Another direction would be to extend the machine learning, such as having more data, more computational power, and more time, to perform more hyperparameter tuning and optimization of the models so that the under- and overfitting could be better balanced. A third alternative could be to compare more machine learning methods, such as SVM with a custom kernel or other type of regression models.
REFERENCES


