Machine Learning for Predictive Resource Scaling of Microservices on Kubernetes Platforms

Adam Rubak
Karlstad University, Karlstad
Sweden
adam.rubak@gmail.com

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

ABSTRACT
Resource scaling is the process of adjusting the amount of resources allocated to a system or a service according to the changing demand. For microservices, resource scaling can be done at different levels, such as the container, the pod, or the cluster. However, the current approaches for resource scaling are not good enough because they rely on reactive or rule-based methods that do not account for the dynamic and complex nature of microservices. These methods often lead to over-provisioning or under-provisioning of resources, both affecting the quality of service and the cost efficiency. To address these issues, this work focuses on testing multiple machine learning approaches to optimize the pod dimensioning problem for Kubernetes platforms through predicting resource requirements for an upscaled number of users. The proposed approach aims to address the limitations of the standard Horizontal Pod Autoscaler (HPA), which often results in resource wastage or suboptimal performance. The results were promising and demonstrated high precision and performance of multiple ML models to accurately forecast future resource needs.

CCS CONCEPTS
• Computing methodologies → Supervised learning by regression; Neural networks; • Computer systems organization → Cloud computing.

KEYWORDS
Kubernetes, Horizontal Pod Autoscaler, ML

ACM Reference Format:

1 INTRODUCTION
The way complex systems are developed and delivered has been transformed by microservices, an architectural paradigm that involves small independent services that communicate through specific APIs [1]. Developing, deploying, and scaling each service independently enables a dependable and speedy system deployment.

Containerization is essential to facilitate the success of microservices. It refers to encapsulating an application and its dependencies into a standardized unit that can be effortlessly executed on any platform. This technique presents multiple advantages, such as isolation, portability, consistency, and efficiency, which simplifies the development and deployment of microservices. Containers offer a streamlined approach to software development by limiting complexity and reducing dependencies through the encapsulation of only essential components and libraries [12].

Despite the benefits of utilizing microservices and containerization, the management and orchestration of containers that execute microservices present notable challenges. To mitigate these complexities, Kubernetes (an open-source platform) offers a comprehensive solution for deploying, scaling, and managing containerized applications. Specifically designed to cater to microservices, Kubernetes streamlines critical functionalities including service discovery, load balancing, self-healing, declarative configuration, multi-cloud compatibility, secrets management, and zero-downtime deployments [12].

Efficient provisioning of service, however, exceeds the provided capabilities of Kubernetes platforms. Anticipating the workload and resource demand of a service and adapting them in real time to circumvent the issues of over-provisioning or under-provisioning of resources is a vital concern in cloud computing. Over-provisioning signifies the provision of resources in excess of what is required, leading to wastage and inefficiency. On the other hand, under-provisioning may result in inadequate resources, thereby causing poor performance and violations of the service level [9].

In order to effectively tackle this challenge, it is imperative for cloud providers to diligently monitor the workload and resource utilization of each service. This can be achieved through the implementation of sophisticated auto-scaling mechanisms that operate in reactive or proactive manners. Reactive autoscaling responds to current or recent workload and resource metrics to dynamically scale the service accordingly. Proactive autoscaling relies on historical data, trends, patterns, or external factors to predict future workload and resource demand, allowing the service to be scaled in advance to meet expected demands [2].
Both reactive and proactive approaches to auto-scaling possess their respective merits and demerits. Reactive auto-scaling is relatively uncomplicated and steadfast, but it may entail increased latency and expenses due to the postponement of scaling actions. Proactive auto-scaling, on the other hand, is more efficient and prompt, but it may involve greater complexity and uncertainty due to the intricacy of accurately forecasting future workload and resource demands [2].

This study delves into two key areas of investigation: (1) to analyze the viability of utilizing ML methodologies in the reduction of Service-Level Agreement (SLA) violations in microservice-based applications [15]; and (2) to study the impact of over-replication on the behaviour of microservices. Service Level Agreements (SLAs) are formal agreements that outline the service quality standards customers expect from their service providers. SLA violations occur when these standards are not met, leading to dissatisfaction and potential contract breaches [6]. Cloud service providers often adopt a reactive strategy to accommodate the growth of their applications. However, this approach may not align with the service level agreement (SLA) requirements when there is a sudden surge in demand for specific services. In such cases, the autoscaler may not be able to respond swiftly enough to adjust the necessary resources, potentially leading to a breach of the SLA [2]. The use of ML presents a viable approach for the efficient scaling of microservices. This approach involves predicting future service requirements and the consequent modification of available resources to meet the demands. Over-replication, which refers to scaling microservice deployments with more replicas than necessary, poses a critical question on whether the system’s response time and availability can still be improved despite the wasted resources (caused by autoscaler over-replication). By addressing these research inquiries, this paper seeks to contribute to advancing effective and efficient resource management strategies in microservices within cloud computing environments.

2 RELATED WORK

Previous research has explored the application of ML models to optimize resource allocation in container-based applications. Two relevant study trends stand out in this context.

One of the early papers highlights that the issue of dynamic provisioning was already thought of before cloud computing was available [14]. The paper was exploring the multi-tier applications and identified that dynamic provisioning for these applications is a challenge that was not thought when researching single-tier application provisioning. This gave birth to new research discussions and later research on autoscaling began where a paper highlighted the strong advantages of this system [4]. However, researchers quickly began to see some disadvantages and proposed another way of scaling, that is, the proactive scaling method.

An early study has tried to create a proactive scaling method that would work both horizontally and vertically using a machine learning method called Long Short-Term Memory (LSTM). In the study, they have used the Google Online Boutique (GoB) [7] (also known as Hipster Shop) as the main application to run the tests. The LSTM model was trained from a real application data and then integrated into the cluster as the predictor. The used metrics to predict the data using the number of requests and an application latency. The focus mostly relied on predicting the values and then feed these values into the Autoscaler to scale the application ahead of time. Their experiment showed promising results as the application availability was higher, and the latency was kept lower compared to other experiments [5]. Other trends started to emerge with other variations of proactive scaling.

The first trend investigates the use of reinforcement learning (RL) policies to control the elasticity of container-based applications. Model-free and model-based solutions were proposed, leading to the creation of Elastic Docker Swarm (EDS) [10]. The model-free methods focus on learning (using an RL method) and makes the model assume nothing about its environment. They are suitable to solve complex issues, however this can take a tremendous amount of time to learn. The learning process can be decreased by using a model-based approach, which focuses mainly on planning, and introduces the learner with basic knowledge of the environment to further simulate the outcome [11, 10]. These two methods, joined together with Dyna-Q algorithm, make what researchers in the study call EDS. The advantages and flexibility of both reinforcement learning approaches were demonstrated, where the model successfully learned the best adaptation policy based on user-defined deployment goals [10].

The second trend investigates proactive scaling engines to leverage multiple AI-based forecast methods to optimize operation in various scenarios. In one study, HPA+ was designed/tested on Kubernetes [13]. Here, the authors introduced an excess parameter to control the resource allocation policy to significantly improve the auto-scaling engine by rejecting fewer requests (while only slightly increasing resource usage).

3 METHODS

Unlike both trends, and in comparison to them, this paper takes a different approach by utilizing supervised learning models for resource dimensioning. In our approach, Kubernetes’ original HPA was used to seed the training data. Different ML models (linear regression, SVM, and MLP) were then used to predict resource needs for upscaled number of users. To be more specific, we used HPA data for a small implementation of a platform (serve a few users) to produce and collect the training data. The collected data is then used to predict resources (CPU, memory, network, and disk) required for running the same system for a much higher number of users.

This study aims to provide a solution to seamlessly scale up the platform that serves a few users to a much larger number of customers. Specifically, Kubernetes’ reactive technique (HPA) will be leveraged to gather essential data concerning resource utilization across various user loads. This data will serve as the foundation for training a machine-learning model to estimate future resource demands based on factors such as user volume, requests per second, and desired average and peak response times. The primary goal is to enable proactive application scaling, ensuring optimal resource distribution and an uninterrupted user experience, even during peak workloads. By assessing the efficiency and effectiveness of our proposed method in enhancing resource utilization and performance scalability within cloud environments, this research seeks to
make significant strides in containerized cloud application resource management.

3.1 The Test Loop

To gain a holistic understanding of the experimental setup, we designed a test loop to automatize the running of our tests, and collect the results (Figure 2). Our platform includes the following software packages: (1) the Google Online Boutique (GoB) to represent the system under test [7], (2) a locust cluster to pressure the GoB to serve/handle different number of users [3], (3) a Kubernetes platform to set the number of replicas for each pod (each microservice) in the GoB application [12], and (4) a Prometheus system to collect all metrics during the operation of the GoB when serving different number of users [8].

GoB is a cloud-native microservices demo application that simulates an e-commerce website. It allows users to browse items, add them to the cart, and purchase them. GoB is designed to showcase the capabilities of Google Cloud Platform, such as Kubernetes, Istio, and Stackdriver.

Locust is an open-source load-testing tool to simulate/define user behaviour with Python code and run distributed load tests with millions of users. Locust supports testing web applications, RESTful APIs, WebSocket, and other types of systems. It is easy to use, scalable, and extensible.

Kubernetes is a portable, extensible, and open-source platform for managing containerized workloads and services. Kubernetes facilitates declarative configuration and automation, as well as service discovery, load balancing, networking, storage, security, and observability. Kubernetes is widely adopted by enterprises and cloud providers as the (de-facto) standard for deploying and operating modern applications.

Prometheus is a powerful monitoring and alerting system that collects and stores time-series data from multiple sources. Prometheus supports a flexible query language to analyse metrics and generate alerts based on predefined rules. Prometheus integrates with many other tools and services in the cloud-native ecosystem, such as Grafana, Alertmanager, and Thanos.

The profiling phase commences by loading a JSON file (Figure 1) encompasses predefined test scenarios to be sequentially executed through the loop. The Locust parameters for each test scenario are sent/transmitted to the Locust API to pressure the system (GoB) with specific loads. The Kubernetes API receives the scaling decision and effectively scales the application (i.e., adjusts the number of replicas for each microservice). Prometheus monitors and records all relevant metrics throughout the specified runtime.

Once a test scenario is concluded, a report is downloaded from Locust and parsed. The Prometheus server is queried to furnish the desired metrics by utilizing timestamps from the Locust report. The information extracted from both Locust and Prometheus is further processed into a structured JSON file, which subsequently undergoes another parsing procedure to generate a convenient data frame suitable for data evaluation and/or ML training. This iterative process handles each provided scenario, ultimately creating CSV and XML files to store the generated data.

The test loop flow proves instrumental in gathering valuable data and assessing the system’s performance under diverse conditions. Leveraging the collected data, ML models are trained, and the efficacy of the proposed approach is rigorously evaluated. Subsequent sections elaborate on this study’s data collection and parsing methodologies.

3.2 Data Gathering

The data-gathering process encompassed three essential components to ensure to gain comprehensive insights into the application’s performance and resource utilization.

3.2.1 Kubernetes. Python scripts were developed to communicate with the Kubernetes API’s RESTful interface to interact with cluster resources and components effortlessly. This facilitated the creation of an automated test loop, essential for generating the necessary data. The data generation process utilizing the Kubernetes API involved the following steps:

- Installation of the Kubernetes-client Python library via pip, which offered a robust Python wrapper for the Kubernetes API, enabling diverse operations on cluster resources and components.
- Development of a Python script comprising distinct classes responsible for various tasks.
  - ‘Scaler’ to adjust the number of replicas.
  - ‘Gatherer’ to collect HPA’s current settings (e.g., CPU threshold).
  - ‘Patcher’ to dynamically updated HPA settings during a test.

3.2.2 Locust. Due to its scalability, flexibility, and ease of use, Locust was employed to evaluate GoB’s performance and response time across various scenarios. It facilitated the simulation of a swarm of users, generating diverse load patterns (e.g., user counts, spawn rates, and test durations) by running multiple Locust instances concurrently. The test results were displayed in a user-friendly web-based interface, presenting pertinent statistics and graphs like requests per second, response time, and failure counts. The interactive interface allowed users to control test parameters, start or stop tests, and gain real-time insights into GoB’s behaviour.

3.2.3 Prometheus and Grafana. a Prometheus server was deployed to gather and store various metrics about the underlying Kubernetes cluster. It collected CPU utilization, network traffic, and memory usage data, all crucial for analysing resource consumption and efficiency. After each test, a report was downloaded for thorough data analysis. Additionally, integrating the Prometheus server with the Grafana dashboard enabled real-time visualization and comparison of metrics across different tests or environments. The Grafana dashboard empowered users to create custom graphs and charts, highlighting specific metrics of interest and revealing relevant trends or patterns. The seamless integration of Prometheus and Grafana provided a comprehensive and insightful overview of GoB’s performance and behaviour, aiding in well-informed decision-making processes.
### 3.3 Data Parsing

This section clarifies the data parsing process, encompassing the utilization of the Locust HTML file and the Prometheus server. The method comprises two fundamental steps: parsing the gathered data and crafting structured machine-learning data frames.

#### 3.3.1 Parsing

Step one involves transforming the collected data into a more amenable format for analysis. The report is downloaded from the Locust web interface to extract crucial information from Locust tests (saved as an HTML file). This file encompasses vital metrics such as response time, user count, failures, and requests per second. To streamline data manipulation and analysis, a custom Python script is devised to parse each Locust HTML file and converting it into a structured JSON file. The JSON file includes timestamps to reflect the start- and finish-time of each test; they are later on used to fetch data from the Prometheus server. By leveraging the Locust JSON file and PromQL language, queries are performed on the Prometheus server to retrieve pertinent data related to CPU usage, network, memory, and disk.

#### 3.3.2 Producing ML data points

In step two, the focus shifts to creating ML data frames, amalgamating data from both Locust and Prometheus sources. The objective is to facilitate ease of use for subsequent normalization, manipulation, and model training tasks. Key metrics, such as response time, CPU usage, and memory usage, are extracted from the data sources and amalgamated with the corresponding number of users. Subsequently, CSV files of the generated data frames are produced and saved, simplifying further data processing and analysis endeavours. These structured data frames are the foundation for developing and training ML models to derive insights and optimize resource allocation for cloud applications.

### 3.4 Finding Best (HPA) Parameters

As explained earlier, the first step is to use Kubernetes’ default HPA to scale pods (microservices) for GoB. Knowing that HPA is not always able to scale up in time to catch up with load bursts (caused by flux of users accessing the application), we are going to use HPA results for a small implementation of the GoB application to learn about its behaviour and how it must be scaled for a much larger number of users. HPA itself, however, has many parameters that need to be properly set before producing acceptable results.

This section is to delve into the crucial process of discovering optimal settings for the HPA governing the scaling of microservices in the GoB application. Emphasis is placed on two vital factors: the stabilization window and the CPU threshold. As observed through multiple experiments, these factors determine the HPA’s responsiveness to fluctuations in CPU usage across pods, and thus significantly influence GoB’s performance.

To identify the most suitable HPA parameters, a comprehensive set of tests was conducted using our test loop; that is, we executed a series of load tests on GoB under varying user loads. Table 1 illustrates the different combinations of stabilization window and CPU threshold values evaluated. Crucial metrics, such as average response time, failures, and requests per second, were recorded for each test. As shown in the table, to find the best set of parameters, the test loop dynamically adjusted the CPU resources allocated to each pod based on the CPU threshold value; for example, when the CPU threshold was set to 40%, each pod received 0.4 of a core (shown as 400 millicores in Kubernetes).

<table>
<thead>
<tr>
<th>User Count</th>
<th>CPU Utilization Threshold (%)</th>
<th>Stabilization Scale Up Window (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>[40, 50, 60, 70, 80]</td>
<td>[15, 30, 45, 60]</td>
</tr>
<tr>
<td>5000</td>
<td>[40, 50, 60, 70, 80]</td>
<td>[15, 30, 45, 60]</td>
</tr>
<tr>
<td>10000</td>
<td>[40, 50, 60, 70, 80]</td>
<td>[15, 30, 45, 60]</td>
</tr>
<tr>
<td>15000</td>
<td>[40, 50, 60, 70, 80]</td>
<td>[15, 30, 45, 60]</td>
</tr>
</tbody>
</table>

To understand the impact of HPA parameter on GoB’s performance, Figures 3 and 4 are provided to demonstrate the average response times under increasing concurrent users. These graphs depict distinct combinations of CPU Utilization Threshold (CUT) and HPA Stabilization Window (SW). CUT signifies the percentage of CPU utilization triggering HPA to scale up or down replicas; SW denotes the duration HPA waits before applying additional scaling actions, mitigating frequent fluctuations.

The findings from these graphs indicate that over-replication may not necessarily enhance GoB performance, that is an actual fact about almost all containerized applications. In some instances, higher response times were observed when pods had lower CPU utilization, contrary to expectations. This highlights the potential increase in resource consumption and network overhead due to over-replication. Notably, the optimal combination of 70% CPU utilization and 45 seconds for SW exhibited the most stable and desirable results, delivering the lowest average response time and minimal variations for up to 10,000 users. We will use these settings during comprehensive profiling stages. Due to practical limitation and the number of servers/cores we had to test our algorithms, we ran tests to gather ML-data for up to 2000 users, and test our ML models to predict GoB performance for up to 10000 users.

### 4 EXPERIMENTAL SETUP

This section presents the initial results of the experiments conducted on the Kubernetes system to evaluate the predictions of ML
We used four ML models: Support Vector Machine with linear kernel (SVM-Linear), SVM with polynomial kernel (SVM-Poly), Multilayer Perceptron (MLP), and Linear Regression (Linear-Reg). The data collected for evaluation comprised CPU usage information from all 11 pods of GoB: adservice, cartservice, checkoutservice, currencyservice, emailservice, frontend, paymentservice, redis-cart, shippingservice, productcatalogservice, and recommendation service.

The ML data was divided into training and testing sets, with the testing set solely employed for assessing the models’ accuracy. A grid search was conducted to identify the most accurate models, and the best-performing ones were selected for further evaluation. The final assessment utilized data collected from tests executed with the highest overall accuracy model. This procedure had three distinct stages:

1. Baseline data was procured from HPA to establish the foundation for further investigation (tests, data gathering, etc.)
2. Creating novel test scenarios (based on the data obtained from the former stage) to modify the requisite quantity of replicas for each deployment. Four scenarios were designed for each entry (44 in total) by randomly manipulating the HPA suggested number of replicas for each pod by ±20%.
3. The novel test scenarios were executed to build and evaluate the ML models. A separate model (268 in total) was trained for every deployment and metric.

5 RESULTS AND DISCUSSIONS

This section reports the findings of using ML prediction models. Here, we used data from experiments with up to 2000 users to train our ML models, and estimate (extrapolated in this case) the future resource demand for users between 2000 and 10000. A technical limitation (number of available CPUs in our private cloud) prevented us from running experiments with more than 10000 users. It is noteworthy to mention that experiments with 10000 users consumed almost 90 cores for GoB, Locust, and Prometheus.

5.1 Resource Prediction per Pod

Figures 5 and 6 show the performance (accuracy) of different ML models. Figure 5 depicts the CPU cores used by each pod and model at varying user loads, displaying both the actual and predicted values. Meanwhile, Figure 6 presents the error rates for each pod and model at different user loads.

Based on Figure 6, it can be observed that certain models outperformed others in predicting the CPU cores required for each pod. Notably, models with linear prediction lines exhibited lower errors across all deployments. MLP with polynomial kernel, on the other hand, led to relatively high errors, particularly for the frontend...
The MLP demonstrated effective capture of the overall CPU usage trends for the checkout service and email service pods. It, however, slightly missed the precise number of cores required by 0.2 – 0.6 cores.

To assess model performance quantitatively, evaluation metrics such as mean square error (MSE) and 95th percentile error were considered. The error, calculated as the difference between the actual and predicted values, was used to derive these metrics.

The results presented in Figure 7 and Table 2 indicate low MSE values for almost all models and deployments, except for the frontend pod with SVM-Poly. The SVM-Linear model demonstrated the smallest variance in errors, implying superior performance compared to other models.

While the MLP exhibited a high MSE of 1.12 for the currency service pod at approximately 10,000 users, the errors for all models were generally insignificant, with the highest being approximately 2 cores more-or-less than the actual value set by the HPA. The lowest error value, around 1.5 cores, was observed for SVM-Linear in the case of the product catalogservice pod at approximately 9,000 users.

### 5.2 Resource Prediction for GoB as a whole

This section focuses on the results of using different ML models to predict the total CPU cores required for the GoB as a whole (i.e., for all its 11 microservices). By aggregating the CPU cores needed for each pod, the total CPU cores for the cluster were calculated and compared with the actual CPU cores set by HPA in its steady-state.

Figure 8 illustrates the cluster’s predicted and actual CPU cores using four ML models: SVM-Linear, SVM-Poly, MLP, and Linear-Reg. The results indicate that SVM-Poly performed poorly, significantly overestimating the CPU cores required. The other three models displayed good performance, closely matching the actual CPU cores.

Table 3 further demonstrates that the MLP outperformed the other models, yielding the lowest variance in errors. This finding suggests that ML models, particularly the MLP, can be effective tools for accurately predicting the total CPU cores required for entire applications (GoB in our case), paving the way for more efficient and scalable cloud applications.

### 5.3 Discussion

The obtained results present a promising step towards harnessing ML for predicting future resource requirements based on historical data. Tables 3 and 2, along with the provided figures, showcase how ML models could accurately predict actual resource requirements for microservice based applications.

However, before these models can be deployed in production environments, addressing their limitations and challenges is crucial. One significant challenge is the concept of extrapolation, where predictions extend beyond the available data range. Extrapolation can be risky, as trends might change over time due to various factors such as evolving user preferences, competitor actions, and seasonal fluctuations. For instance, predicting product demand based on past sales data might be effective, but unforeseen shifts in consumer behaviour or market dynamics might not be captured. Additionally, the applicability of these models across diverse domains and scenarios requires careful consideration, ensuring alignment with specific use cases.

The issue of extrapolation is especially pertinent to this work. While the models show promising results in forecasting future resource needs, it is essential to recognize the potential instability of extrapolation, which could lead to system malfunctions. For example, in high-stakes applications (such as medical systems or autonomous driving) incorrect predictions could have severe consequences, necessitating cautious deployment. On the other hand, low-risk domains (such as entertainment or social media) might find utility in extrapolation for enhancing user experience without catastrophic implications for occasional inaccuracies.
Regarding this paper’s scope, it is essential to remain mindful of the implications of extrapolation. The proposed solution can be valuable for applications seeking resource optimization based on user count predictions. However, its use in critical systems must be cautiously approached, prioritizing consistent and reliable performance. Another aspect to consider is the size of the datasets used in this study. With relatively limited data points, accurately discerning overarching trends becomes challenging. ML models might be misled by such small datasets, necessitating the exploration of more extensive data collection to gain comprehensive insights into system behaviour.
In parallel and despite its promising outcomes, it is also essential to recognize that ML models (in general and including ours in this paper) can predict future resource needs, but they do not guarantee the complete elimination of SLA violations. This leaves room for improvement and further investigation. Therefore, future work could focus on enhancing model accuracy and robustness by refining models with additional features and parameters, and exploring advanced ML techniques (such as deep learning, reinforcement learning, or generative adversarial networks) for novel insights and solutions.

6 CONCLUSION

This research focused on utilizing ML methods to predict the future resource requirements of a Kubernetes system based on collected data gathered for a system when serving a fewer number of users. Our study involved a comprehensive process, including a literature review, setting up a testing environment, conducting experiments, optimizing the Kubernetes’s native HPA variables, and using various ML models to properly scale-up (dimension) various underlying microservices.

Data collection involved running tests with various scenarios, followed by training ML models with the collected data. The validation results demonstrated that although none of the ML models could precisely estimate the exact resource needs, some were very close. Through our experiments, we also showed that incorrect HPA settings in Kubernetes could lead to resource over-provisioning or under-provisioning, to which, our proposed ML models were designed to offer a proactive solution for scaling efficiency.

Several potential directions for future research could enhance the findings and expand the scope of this study. Firstly, gathering more data and increasing the number of concurrent simulated users could reveal additional insights and patterns that influence system performance and prediction accuracy. Secondly, leveraging more advanced ML techniques (e.g., reinforcement learning) could lead to dynamic system adjustments based on real-time data and feedback. This dynamic approach could allow the system to continuously optimize itself, making it more adaptable to changing user demands and scenarios.

REFERENCES


Received 2020 February 7; revised 12 March 2009; accepted 5 June 2009