Pod Autoscaling in Kubernetes Cluster

Master thesis by Angelina Horn

Date of submission: December 27, 2020

1. Review: Prof. Dr. Felix Wolf
2. Review: Dr. Hamid Mohammadi Fard
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Parallel Programming

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Angelina Horn, die vorliegende Masterarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 27. Dezember 2020

Angelina Horn

Contents

1 Introduction

1.1 Motivation

Lately microservice architecture is gaining popularity especially among developers [1]. The microservice architecture describes a new and highly flexible way of application architecture and deployment. For this approach an application gets split into smaller but independent chunks called microservices. The created microservices will be deployed, for example as docker containers in a server cluster. These clusters will often not be maintained or owned by the applications developer but rather by an enterprise cloud provider like Google or Amazon. Microservices that belong to one deployment can communicate between one another via lightweight API interfaces e.g. REST.

Applications will have a dynamic load based on several possible reasons such as daytime and workload [2]. This load however is not distributed equally among all microservices. Therefore, it is necessary to be able to scale the microservices while they are under load in order to maintain performance of the whole application. Additionally, because of limited and costly resources, one wants to use only the minimal amount of resources that are necessary to sustain performance.

This can be achieved via automatic scaling [3]. Automatic scaling describes a mechanism which periodically monitors every resource of a microservice, determines if and how a container should be scaled based on metrics defined by the developer and executes the scaling automatically. Because all these described architectures and ways of deployment are rather new there is room for research and improvement regarding this topic.

1.2 Objective

The objective of this thesis is mainly to implement a machine learning process as part of a fully automatic process, that produces a performance model of a docker container application, which on one hand determines the scaling technique (vertically, horizontally or both) and on the other hand how much and when to scale with the goal of not exceeding a given maximum response time.
Therefore the implementation can be divided into three separated goals (see **??**):

1. The design of a graphical user interface which will be the single interaction point of the user.
2. The development of a machine learning algorithm which takes performance data of a microservice as an input and produces a performance model of this microservice as a result.
3. The implementation of a control script which connects the graphical user interface, the machine learning algorithm and Kubernetes.

The combination of these three goals will result in the above described fully automated process. Which will start with the input of the microservice docker image from the user and result in a fully deployed and auto-scaled microservice in a given Kubernetes cluster.

2 Background

In this chapter of the thesis, basic topics of which their understanding is necessary, will be pointed out and explained. These topics are Microservices, Kubernetes and Autoscaling. Furthermore two applications that play an important role in this thesis called OctoScan and Extra-P will be presented. These topics will build the foundation and background needed for in this thesis following approaches, research and statements.

2.1 Microservices

The term ”microservice” was first introduced by James Lewis in a conference [4]. It describes an application architecture which is characterized by subdivided applications, that communicate with each other trough lightweight interfaces.
Each of these applications has a single responsibility or task and is deployed independently. This means that each of these microservices can be implemented in a different programming language and can be deployed on an entirely different operating system on a different server as long as they share a common interfaces and are able to communicate with each other trough it. James Lewis describes this approach as ”[...] rooted in the the Unix Philosophy of small and simple” [4].

On one hand this architecture offers several advantages compared to the monolithic application architecture which was popular prior. First and foremost it provides modularity which is highly beneficial regarding code complexity as well as team coordination [5]. The introduction phase of new programmers is highly decreased as well as the possibilities to work on different parts of the overall application at the same time.
Secondly, this kind of architecture makes it possible to incrementally upgrade an application and therefore possibly to reduce time to market [5]. Certain parts of the application can be updated independently from the other microservices regarding not only the application code itself but also its dependencies such as libraries and environments. This way the so called ”dependency hell” [6] which many monolithic applications suffer from can be avoided.
Furthermore microservices are more resistant to failure due to their containerized existence. If one microservice fails it does not necessary lead to the failure of the whole application. Additionally this failed microservice can be easily restarted in a short amount of time because of its lightweight nature.
Last but not least, microservices introduce a new capability of scaling [5], [7]. Regarding the application, each microservice is under inconsistent load and can be scaled individually. This allows optimal usage of resources in dependency of desired performance and costs.

On the other hand this architecture brings certain disadvantages. One major disadvantage is the time it costs to convert any application in several microservices [5]. This time should not be underestimated but can also bring major benefits as stated above. Another disadvantage is the complexity of the deployment [5]. Today there are several ways to deploy microservices including the use of Jenkins[1] or lately the use of orchestral cluster tools such as Docker Swarm[2] and Kubernetes[3].

2.2 Kubernetes

The term Kubernetes gained a lot of popularity over the past five years (see figure **??**). It is the name of a orchestral tool for containerized applications, originally designed by Google and now owned by the Cloud Native Computer Foundation[4]. Its capabilities include deployment, scaling and monitoring of containerized microservices in clusters. These clusters can be hosted in a public or private cloud.

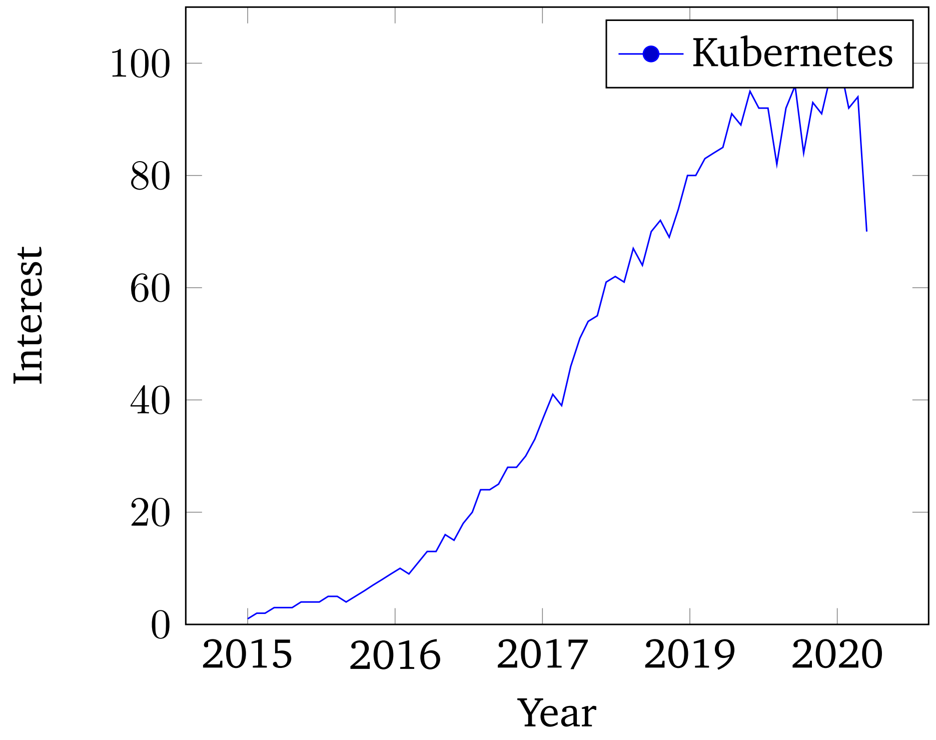


Figure 2.1: Google Trends report for the keyword Kubernetes [8]

Deployment Containerized microservices are mostly available as so called docker images. The configuration of an application based on several microservices gets described in a ”yaml” file which can be read by Kubernetes. In it several aspects of the deployment can be configured - examples are the ports on which the microservice is reachable, the number of replicas that should be deployed or the application to which the microservice belongs. Once this configuration files is made the deployment of this particular microservice is automated.

[1]https://www.jenkins.io
[2]https://docs.docker.com/engine/swarm
[3]https://kubernetes.io
[4]https://www.cncf.io

Scaling There are two general ways to realize the scaling of a microservice. The first one is to scale a microservice manually by for example defining the resources that are available to an application or define the amount of replicas in its configuration file. The second and more complex method is the automatic al scaling of microservices.

Monitoring In order to provide stability of an application it is necessary to monitor its behavior. This enables the discovery of failures, resource use and load on the individual microservices of the application. This information is important to maintain an application. Kubernetes offers several automatic services to support the maintenance. One of them is the capability to restart failed microservices according to certain settings automatically.

A Kubernetes cluster consists of two main structures (see figure 2.2). The first one is called ”Master” and contains the API interface, the controller manager, the scheduler and a configuration storage called ”etcd”. The API interface is the connection point between the developer and Kubernetes. The controller manager is the executing unit of it and the scheduler plans its actions.
The second important structure of a Kubernetes cluster is the node. In contrary to the master there can be several nodes in a cluster. Each node contains several Pods. Each pod represents one microservice. Furthermore each pod contains a ”Kubelet” which is the executing unit of a node similar to the controller manager in the master. Additional the ”cAdvisor” unit monitors the pods of a node and the ”Kube-Proxy” is the connection point between a pod and the users.

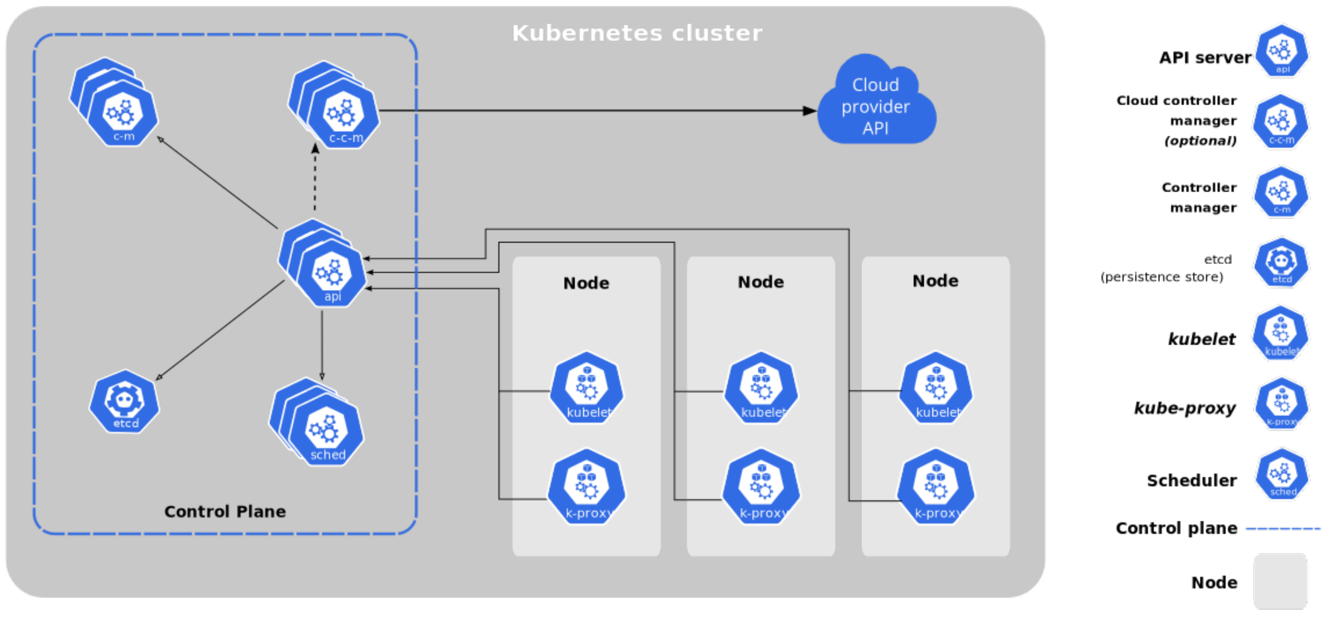


Figure 2.2: Kubernetes cluster overview [9]

Kubernetes offers a very powerful tool for the life cycle of an application in a cluster and paths the way for extensive development of more efficient methods to develop and maintain applications.

2.3 Autoscaling

The term Autoscaling describes the automatic resource scaling of applications and more detailed microservices [10], [11]. As described in the chapter about Kubernetes: autoscaling represents the counter part to the traditional manual scaling of applications. With this method it is possible to allocate more resources for one microservice when it is under especially heavy load or allocate fewer resources when the load gets easier depending on its degree of use in the application by the user.There are in general two types of scaling methods [10], [11]:

Horizontal A microservice gets scaled horizontally when its number of instances gets increased or decreased.

Vertical A microservice gets scaled vertically when the resources of one instance are being increased or decreased.

These two methods describe how a microservice gets scaled but it remains to know when to scale a microservice or the scaling timing. There are existing several approaches: reactive, proactive or even predicted. Furthermore there are several other aspects to pay attention to for example scaling indicators, resource estimation, adaptivity and many more (see figure 2.3)

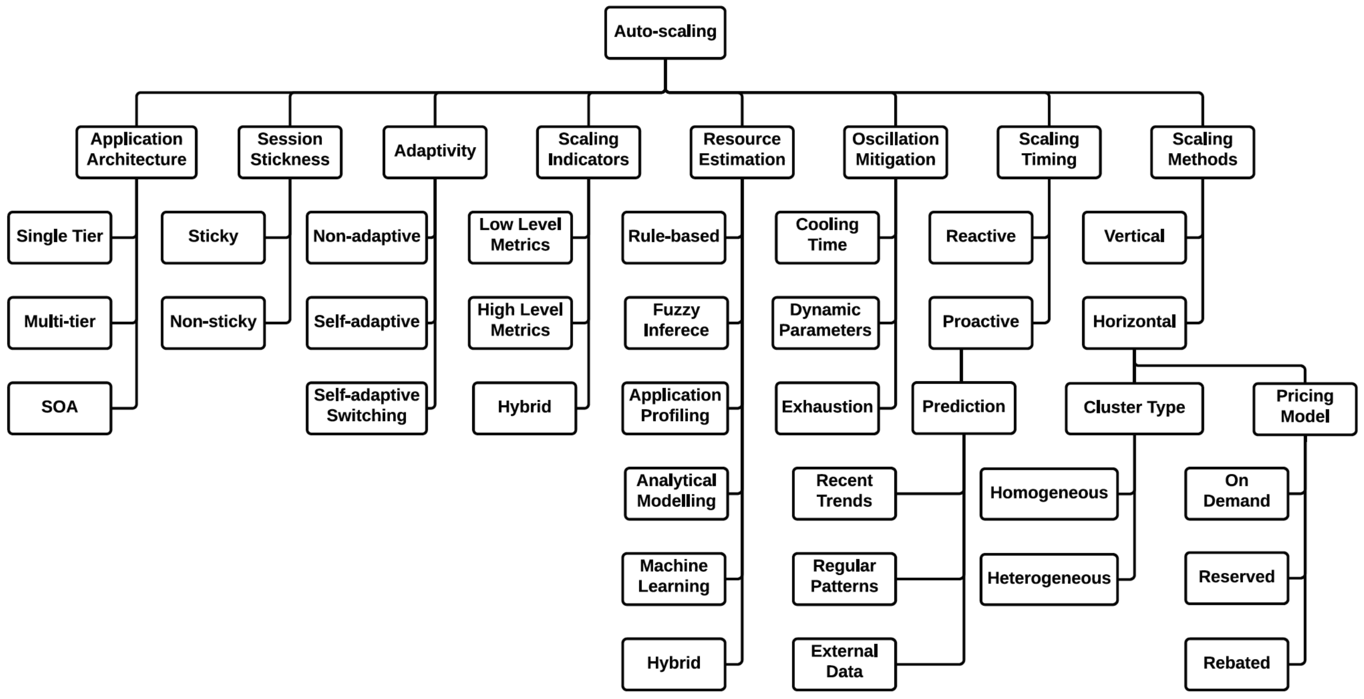
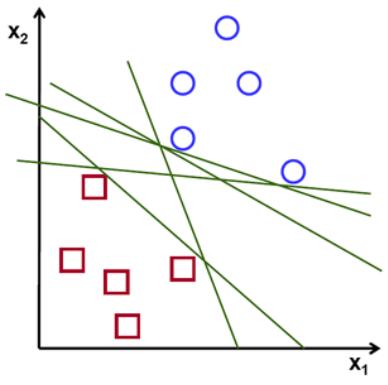


Figure 2.3: Autoscaling taxonomy [10]

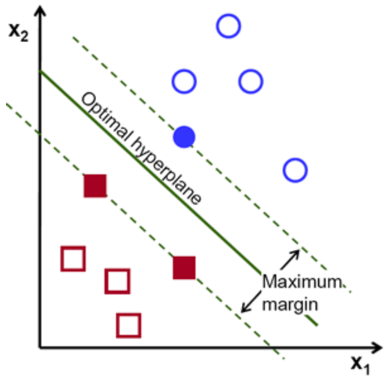
The easiest and most common approach to handle all these factors is to define a threshold for certain resources and a scaling method for each microservice manually. When the resource usage of a microservice reaches the set threshold it will scale according to its set configuration.
Another way is to use a machine learning algorithm which will be trained to know when and how to scale a certain microservice. This approach will be researched further and implemented in this thesis.

2.4 Support Vector Machines

The Support Vector Machine theory was introduced by Vapnik of the AT&T Bell Laboratories in 1995 [12]. It describes a supervised machine learning model which is used for classification and regression.
The SVM algorithm is designed to find a hyperplane in a n-dimensional space which distinguishes data points [13]. In general there are several hyperplanes that are possible, but this algorithm picks the hyperplane with the constraint such that it provides the maximal distance between itself and each class (see figure 2.4). This distance is also called margin, while the nearest data points from each class that determine the maximal margin and therefore influence the hyperplane are called support vectors. The dimension of the hyperplane depends on the number of features R^n. A hyperplane in for example R^2 represents a line while it represents a two-dimensional plane in R^3 [13].
In order to handle non-linear problems it uses the kernel trick to transfer the function into higher dimensional



(a) Possible Hyperplanes



(b) Hyperplane with maximum margin

Figure 2.4: SVM Hyperplanes [14]

feature space [15]. Therefore the used kernel function influences the performance of the SVM algorithm significantly and should hence be chosen wisely. The most common kernel functions are ”polynomials, radial basis functions and certain sigmoid functions” [15].
Support Vector Machines can, as mentioned before, also be used for regression. In order to realize this, it is

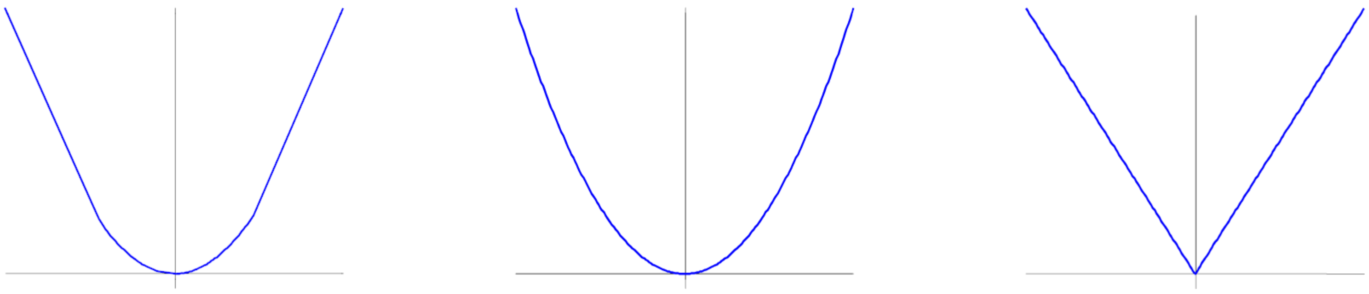


Figure 2.5: SVR loss functions [15]

necessary to introduce a loss function that includes a distance measure [15]. Possible loss functions are for example Quadratic, Laplace or Huber functions (see figure 2.5).

3 Related Work

4 Method

4.1 Overview

4.2 Environment Setup

4.3 Implementation

GUI

Performance Model Selection of parameters * CPU utilization * CPU share * Memory utilization * Memory share * responds time

Machine Learning Model * Support Vector Machine

Control Script

4.4 Extra-P

4.5 Experimentation Methodology

5 Evaluation

5.1 Results

5.2 Analysis

6 Conclusion

6.1 Summary

6.2 Outlook

References

[1] N. Alshuqayran, N. Ali, and R. Evans, “A systematic mapping study in microservice architecture”, in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, 2016, pp. 44–51.

[2] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”, in *2015 10th Computing Colombian Conference (10CCC)*, IEEE, 2015, pp. 583–590.

[3] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning”, in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019, pp. 329–338.

[4] J. Lewis, “Microservices-java, the unix way”, in *Proceedings of the 33rd Degree Conference for Java Masters*, 2012.

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: yesterday, today, and tomorrow”, in *Present and ulterior software engineering*, Springer, 2017, pp. 195–216.

[6] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment”, *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[7] D. Namiot and M. Sneps-Sneppe, “On micro-services architecture”, *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[8] (Oct. 2020). “Google Trends”, Google LLC, [Online]. Available: `https://trends.google.com`.

[9] (Nov. 2020). “Kubernetes Documentation”, Kubernetes, [Online]. Available: `https://kubernetes.io/docs/`.

[10] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-scaling web applications in clouds: A taxonomy and survey”, *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[11] D. Midigudla, *Performance Analysis of the Impact of Vertical Scaling on Application Containerized with Docker: Kubernetes on Amazon Web Services-EC2*, 2019.

[12] V. N. Vapnik, “The Nature of Statistical Learning Theory”, *New York: Springer Verlag*, 1995.

[13] M. Awad and R. Khanna, *Efficient learning machines: theories, concepts, and applications for engineers and system designers*. Springer Nature, 2015.

[14] Y. Lin, H. Yu, F. Wan, and T. Xu, “Research on classification of Chinese text data based on SVM”, in *IOP Conference Series: Materials Science and Engineering*, 2017, pp. 1–5.

[15] S. R. Gunn *et al.*, “Support vector machines for classification and regression”, *ISIS technical report*, vol. 14, no. 1, pp. 5–16, 1998.