Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor’s Thesis

Automatic resource scaling in cloud applications - Case study in cooperation with AEB SE

Simon Weiler

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Floriment Klinaku, M.Sc. Dipl. Inf. Benjamin Jung, AEB SE
Commenced:	December 1, 2020
Completed:	June 1, 2021

Abstract

As an increasing number of applications continue to migrate into the cloud, the implementation of automatic scaling for computing resources to meet service-level objectives in a dynamic load environment is becoming a common challenge for software developers. To research how this problem can be tackled in practice, a state-of-the-art auto-scaling solution was developed and implemented in cooperation with AEB SE as a part of their application migration to a new Kubernetes cluster. Requirement elicitation was done via interviews with their development and IT operations staff, who put a strong focus on fast response times for automated requests as the main performance goal, with CPU, memory and response times being the most commonly used performance indicators for their systems. Using the collected knowledge, a scaling architecture was developed using their existing performance monitoring tools and Kubernetes' own Horizontal Pod Autoscaler, with a special adapter used for communicating the metrics between the two components. The system was tested on a deployment of AEB's test product using three different scaling approaches, using CPU utilization, JVM Memory usage and response time quantiles respectively. Evaluation results show that scaling approaches based on CPU utilization and memory usage are highly dependent on the type of requests and the implementation of the tested application, while response time-based scaling provides a more aggregated view on system performance and also reflects the actions of the scaler in its metrics. Overall though, the resulting performance was mostly the same for all scaling approaches, showing that the described architecture works in practice, but a more elaborate evaluation on a larger scale in a more optimized cluster would be needed to clearly distinguish between performances of different scaling strategies in a production environment.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction to auto-scaling

When a company or an individual wants to provide an application to a big number of users, it is commonly hosted as a service on one or multiple web-servers that can then be accessed over a network. Depending on the context of use and number of users, an estimation can be made beforehand to predict how many resources and machines are going to be needed to provide a stable and responsive service for all clients. For smaller applications, this approach is sufficient most of the time, even when the performance requirements change, as additional units can be added to scale the service out during high demand or, on the contrary, be removed to scale it in during low demand periods. If these scaling actions are not performed by a human but instead automatically by an intelligent system, the process can be classified as “auto-scaling”. Herbst et al. [HKR13] further elaborate on the latter, defining the term “elasticity”, which is the ability of a system to adapt itself to changing workloads by adding or removing resources to match the given demand as closely as possible. With the ever increasing size and complexity of applications, coupled with rising requirements regarding availability and responsiveness in a dynamic internet landscape, maximizing the elasticity of a system has become a growing challenge for the industry and an exciting field for new research and innovation.

1.2 Problem statement / Case study context

The main focus of this research is observing the introduction of auto-scaling in a practical context by conducting a case study in cooperation with AEB SE. The company, which was founded in 1979 and has around 500 employees, is a software development company located in Stuttgart, whose focus is developing and maintaining applications for managing customs processes, trade compliance and intralogistics & supply chain.

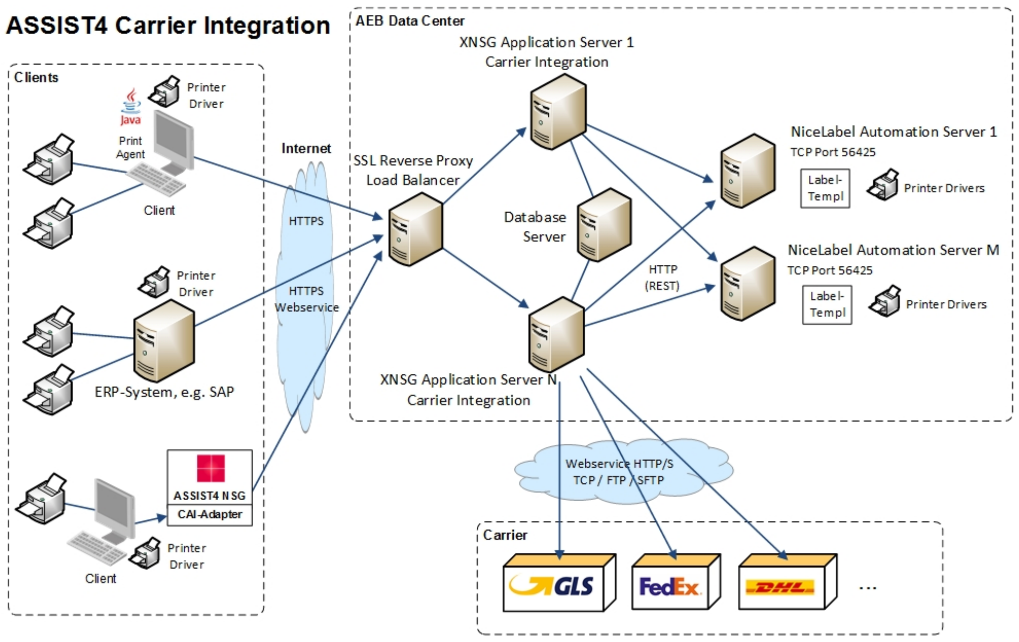


Figure 1.1: Architecture of AEB's Carrier Integration Service

To illustrate AEB's case, Figure 1.1 shows a simplified architecture of their Carrier Integration Service and how it is currently operated. Requests arrive from clients via HTTPS and are first received by a load balancer, which then spreads them across an array of identical application servers. These servers are also connected to other components like a database or external services to provide additional functionality. Running on the servers are Windows virtual machines, hosting the application itself, and an additional "agent" software responsible for managing the installation. While the agent can perform small maintenance tasks on it's own, the bulk of the server management is performed manually by a team of system admins.
The admins' set of tasks therefore also includes ensuring that enough system resources are available to process the workload and to perform scaling operations when needed. While this is an approach that has worked well for many years, its flaws become more apparent as the overall demand increases and consequently fluctuates more. On the occurrence of a sudden spike in requests, the admins need to become aware of the problem first, which usually occurs rather late, as alerting systems on the servers only trigger once they are already processing the requests, leading to a critical delay that can create a small window in which the servers are overloaded and potentially vulnerable to failure. After the load demand has been signaled though, the admins need to manually start and setup additional application servers to handle the incoming wave of requests, creating a further delay. On the other hand, once the high workload has been fully processed and demand returns to normal, the additional, potentially idle, instances now also need to be shut down again manually.

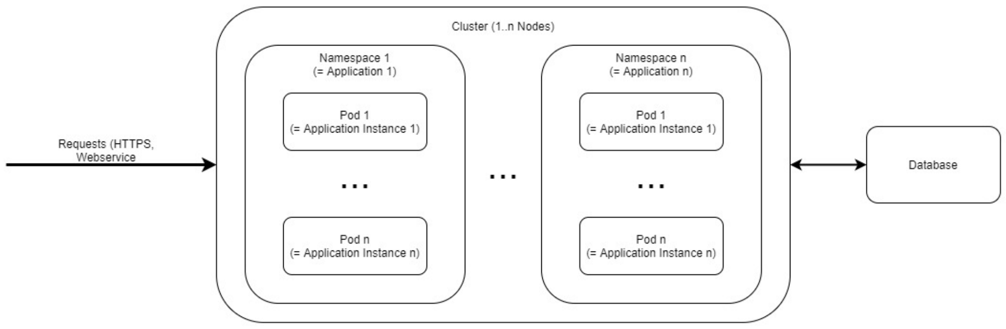

Cluster (1..n Nodes)
Namespace 1
(= Application 1)
Pod 1
(= Application Instance 1)
...
Pod n
(= Application Instance n)
...
Namespace n
(= Application n)
Pod 1
(= Application Instance 1)
...
Pod n
(= Application Instance n)
Requests (HTTPS, Webservice
Database

Figure 1.2: New architecture using Kubernetes

Due to these mentioned issues, demands for a modern and more effective scaling solution came up, which has been in development at the company for a few years now. The new architecture shown in Figure 1.2 replaces the old array of servers with a Kubernetes cluster hosting a number of separate namespaces, each containing a different AEB application. Apart from the benefits of a much more efficient and automated server management thanks to the tools provided by Kubernetes, this new approach now assigns a variable number of pods containing the application instances to each namespace. This pod structure not only provides a faster and easier to manage alternative to regular VMs, but also allows for the implementation of a fully automatic scaling solution, which will be the core interest of research in this case study.

1.3 Research objectives

With AEB's new auto-scaling solution using Kubernetes being in active development, there are multiple topics which provide important and interesting research questions, both on the practical side, in terms of defining and implementing the new system, and on the research side, by looking at how auto-scaling can be introduced efficiently in practice and what the requirements are. Conclusively, this thesis has three main objectives:

1. As this is being developed in a practical setting and on a sizable scale, there needs to be a clear set of requirements and performance demands for the auto-scaling system. Additionally, there should be an evaluation on how precisely the desired results can actually be defined and how they can be altered and respectively configured in the future.

2. To trigger certain scaling operations, one first needs to analyze the systems in use and determine if there is a need for action and in which direction, being either a scale-out or a scale-in operation and at which step size. This requires a technical analysis of the application load, both on the side of the system and its utilization and on the client's side based on the volume of requests. On top of these technical measures, it is also important to look at the expert knowledge present at the company, as many people like developers and architects have a deeper understanding of the systems and their properties, that can help in evaluating or predicting certain load scenarios.

3. Finally, after a base of requirements and expertise has been established, the auto-scaling mechanisms themselves need to be implemented. Here there will be a focus on the different scaling rules that can be defined using the previously collected metrics and how they perform against each other in certain testing environments. Additionally, the basic scaling implementation of Kubernetes will likely not be sufficient, so it needs to be looked into how state of the art scaling techniques, both from research and other production systems, can be incorporated and used for this specific instance given the circumstances and available technologies.

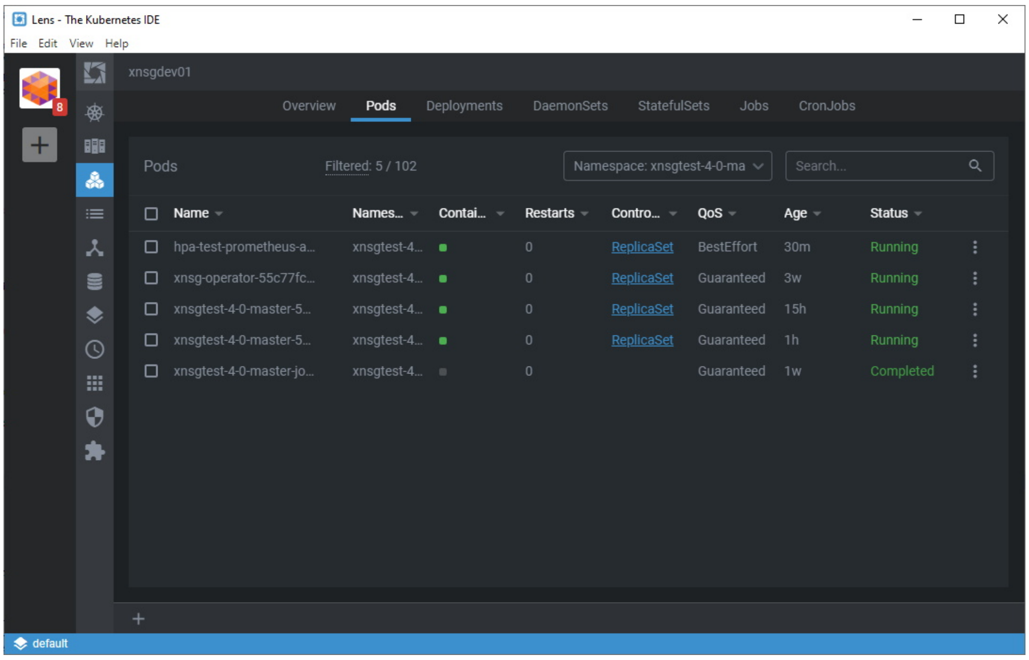
2 Foundations

2.1 Kubernetes (K8s)

2.1.1 General

The first and most integral technology of this study is Kubernetes [K8S] (stylized as K8s for short), setting the groundwork and providing some of the tools necessary to make the scaling implementations possible.
At its' core, Kubernetes is a management and deployment framework for containerized applications that eases many aspects of automatic deployment and scaling by relying on certain key concepts and standardizing many aspects of traditional deployment. It was originally developed at Google and has been available to the public as open-source since 2014. Since then, it has become widely used for elastic deployment, especially in cloud environments, which is also where it is used at AEB as explained in section 1.2.
The main structure used in Kubernetes is a cluster, which contains every other unit and defines the overall framework of the system. On a machine level, a cluster contains a set of physical nodes which represent the actual servers running the applications. One of the core strengths of Kubernetes is that the user is not bound to these physical limitations and instead works on a higher level on abstraction, consisting of pods. A pod is a temporary container that hosts instances of an application in a pre-configured environment, usually using fewer resources than a comparable virtual machine. To give the cluster more structure, pods are also usually grouped together in namespaces and deployments, allowing the hosting of multiple disjointed applications inside the same cluster.
Apart from the powerful management and deployment tools provided by the Kubernetes framework, the most important component for this thesis is the Horizontal Pod Autoscaler [HPA]. Once setup for a deployment, the scaling algorithm will add and remove pods over time to ensure certain performance goals given by the user. The standard configuration takes a desired CPU usage value, together with a minimum and maximum amount of pods, and then estimates how many pods need to be running in that interval to keep the average CPU usage across the pods at the desired level.

2.1.2 Examples



Pods Filtered: 5 / 102 Namespace: xnsgtest-4-0-ma

Name	Names...	Contai...	Restarts	Contro...	QoS	Age	Status
hpa-test-prometheus-a...	xnsgtest-4...	🟩	0	ReplicaSet	BestEffort	30m	Running
xnsg-operator-55c77fc...	xnsgtest-4...	🟩	0	ReplicaSet	Guaranteed	3w	Running
xnsgtest-4-0-master-5...	xnsgtest-4...	🟩	0	ReplicaSet	Guaranteed	15h	Running
xnsgtest-4-0-master-5...	xnsgtest-4...	🟩	0	ReplicaSet	Guaranteed	1h	Running
xnsgtest-4-0-master-jo...	xnsgtest-4...	▪	0		Guaranteed	1w	Completed

Figure 2.1: Pods in a K8s namespace displayed in Lens

To illustrate the use of Kubernetes and its' concepts in practice, figure 2.1 shows a screenshot from the cluster management tool Lens [LNS], which allows visualizing and managing the components of a Kubernetes cluster.
The selected namespace "xnsgtest-4-0-master" contains five different pods from three ReplicaSets, a special form of deployment containing a variable number of identical pods. While each pod can hold a variable number of containers inside of it, in this example each pod only contains a single one, equating to one application instance per pod. All pods are actively running with the exception of the bottom one, a temporary pod used for a specific job, which it has already completed as indicated by its' status, so the container inside of it has stopped.
There are also other namespaces running inside the cluster besides the shown one, either for other products or other branches, together with specially dedicated namespaces for core Kubernetes functionality and extensions. For example, the "kube-system" namespace contains objects maintained by the Kubernetes system itself, like the API server used for communicating with and managing the cluster. Another important namespace for this thesis is "metrics", containing pods hosting monitoring applications like Prometheus and Grafana, which are further explained in section 2.2.

2.1.3 Usage

On top of the reasons for implementing Kubernetes at AEB and its' general architecture as described in section 1.2, using Kubernetes as the centerpiece of the scaling concepts explored in this thesis also has multiple benefits.
Firstly, the standardized deployment approach using elements like Pods and ReplicaSets provide a

foundation for developing a scaling solution mostly independent of concrete application logic. This is both useful for sharing the same scaler for multiple different products inside AEB but also allows it to be used for other use cases.
Secondly, the Horizontal Pod Autoscaler provides an optimized and ready-to-use scaling solution that, in its' base form, can already be applied to the cluster without any adjustments. The extensible and flexible Metrics API enables the definition of custom and external metrics not recorded by Kubernetes itself, allowing the scaler to react to any user-defined metric, provided it is made available through an API. Additionally, the scaling behavior itself can be modified and fine tuned, with custom scaling policies for scaling in and out or by defining different cool down periods.
This simple but powerful set of tools provided by Kubernetes and its' Horizontal Pod Autoscaler provide a solid basis for exploring and implementing the different metrics and scaling concepts gathered in this study. Furthermore, it ties into the ongoing expansion of the Kubernetes implementation at AEB by pushing towards a more standardized and automated approach to scaling their applications.

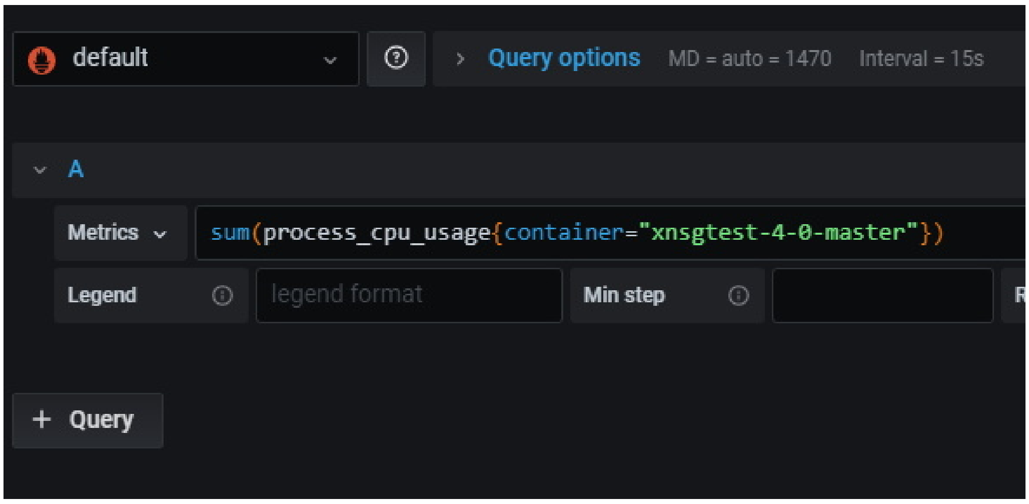
2.2 Prometheus/Grafana

2.2.1 General

The main tools for collecting and working with application and runtime metrics for this thesis are Prometheus [PMS] and Grafana [GFA].
Prometheus is an open-source monitoring and alerting technology for applications. It provides the necessary interfaces for collecting different metrics and performance data from a system, ranging from basic hardware measures like CPU and RAM usage, up to more detailed information like JVM Heap space and even application specific data if set up correctly. To navigate the huge pool of data collected, Prometheus provides its own query language called PromQL [PQL] , which allows for filtering and operating on the raw result data to get the desired insights and graphs.
Due to Prometheus' very limited own visualization, Grafana is commonly used in conjunction with it, as it provides a powerful framework for displaying Prometheus queries and data in an efficient and understandable manner. An instance of Grafana is split up into multiple dashboards, each containing a collection of different panels displaying specific graphs each. When using Prometheus as a data source, each panel contains one or multiple PromQL queries, whose results are then commonly displayed in a two dimensional line chart, with the vertical axis displaying the values returned by the query and the horizontal axis used for a time scale of variable size and precision. By creating a number of these panels and displaying them side by side on a dashboard, Grafana allows for a quick and precise analysis of the current system performance or other relevant metrics.

2.2.2 Examples

One commonly interesting metric for system performance is CPU usage, which can also be collected and analyzed using Prometheus and Grafana. Figure 2.2 shows how this can be accomplished using PromQL.



```
sum(process_cpu_usage{container="xnsgtest-4-0-master"})
```

Figure 2.2: PromQL query for CPU usage

As shown by the highlighted code in the query, it consists of three major parts. The semantically first parameter is the name of the metric “process_cpu_usage”, which defines the data source of the query and is collected via an interface from the processes themselves. Without any modifications, this would result in a huge amount of intersecting graphs for all monitored processes on all containers in the cluster. To narrow this amount of data down, the filter “container=’xnsgtext-4-0-master’” is added, meaning only the data coming from the container with the given name is shown. Finally, since there are two different processes running on the container at the moment, the “sum” operator adds their usage values up to one summarizing curve. The result is shown in figure 2.3.

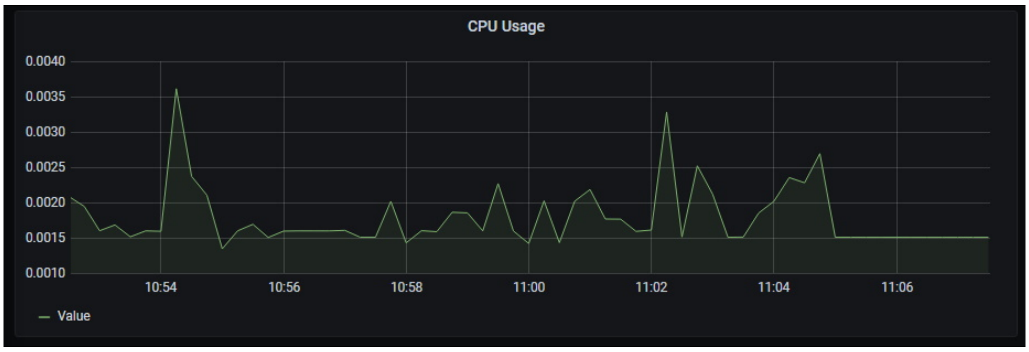


Figure 2.3: Visualized CPU usage in Grafana

The resulting graph shows the desired total CPU usage as an absolute value, on the vertical axis, over the span of the last 15 minutes, on the horizontal axis. Thanks to the visualization, notable data points like spikes and the overall usage tendency are easily visible and analyzed.

2.2.3 Usage

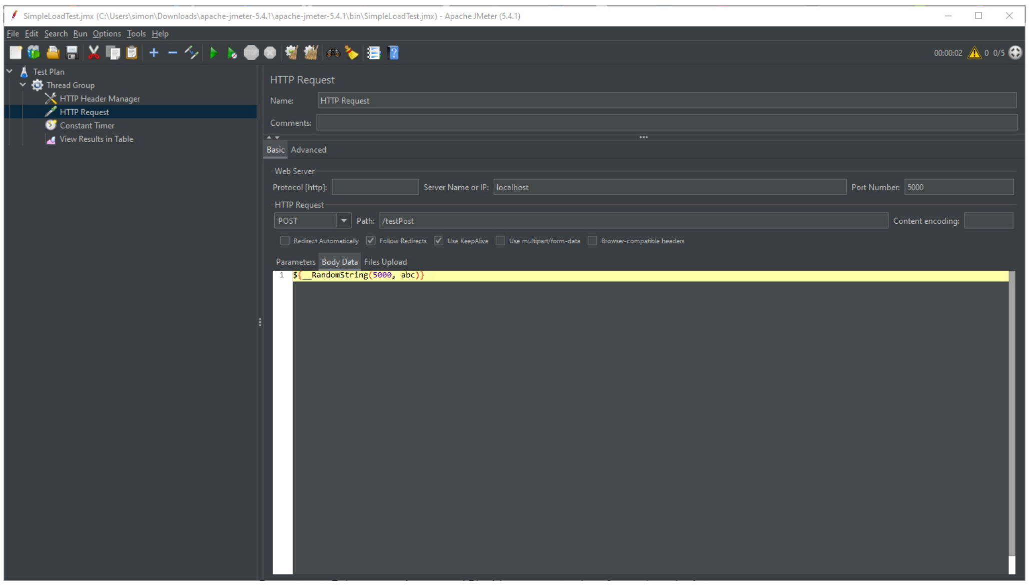
As monitoring tools, Prometheus and Grafana are widely used in the industry, including at AEB, where they are directly linked with most applications and provide valuable performance insights, both for IT operators but also regular developers. Since this infrastructure has already been established, it makes sense to use it as the main way to collect data and metrics for an automated scaler and to evaluate the performance during testing.
For more detailed analysis, Prometheus allows for the definition of very precise queries on performance data, that can be extracted and read by a scaling solution. Additionally, the visualization of Grafana allows for easy experimentation with different metrics to get a feeling for the system behavior and to find out which values are indicative of certain performance properties e.g. which hardware or software metrics closely display the current system usage. On top of choosing the right metrics, defining the right filters and operators to get conclusive and useful results is also an important decision eased by Grafanas visualizations.
Finally, apart from feeding the collected monitoring data into the scaler, using it for manual evaluation is another important ability. The querying tools of Prometheus allow the retrieval of very specific data interesting for certain evaluation aspects, be it low-level performance measurements up to the response time of one concrete web service call, while Grafana's graphs provide an excellent way to visualize the collected data and to graphically compare different results and behaviors.

2.3 Apache JMeter

2.3.1 General

Another technology used in this work is Apache JMeter [JMR], which plays a major role in the evaluation and testing phase.
JMeter is an open-source Java-based tool for configuring and running load tests for web applications. Since its' release in 2006, it has developed a multitude of features and abilities to create and execute very specific usage scenarios to simulate different user behavior patterns. This provides a quick and efficient alternative to manual testing when looking for bugs and errors and a way to create large-scale load testing to put stress on a system and measure performance. Due to it's widespread use, it has a big library of plugins that allow it to be tailored even further to specific applications and to be integrated into other tools like Jenkins for performing automated tests at build time.
Test plans in JMeter are built in a modular fashion, consisting of different modules that each serve their specific function during the test. The first and main component is the Thread Group, which defines the number of threads and thus the overall scale of the test. Inside the group, a multitude of controllers is used to specify the testing behavior, with Samplers containing different types of requests that will be sent to the server and Logic controllers allowing further customization using loops and branching elements. Other commonly used components are Configuration Elements that define the framework of the test, e.g. a list of login credentials to use for user authentication, and Listeners to receive and evaluate the responses from the server.

2.3.2 Examples


SimpleLoadTest.jmx (C:\Users\simon\Downloads\apache-jmeter-5.4.1\apache-jmeter-5.4.1\bin\SimpleLoadTest.jmx) - Apache JMeter (5.4.1)
File Edit Search Run Options Tools Help
00:00:02 0 0/5
Test Plan
Thread Group
HTTP Header Manager
HTTP Request
Constant Timer
View Results in Table
Basic Advanced
Parameters Body Data Files Upload

HTTP Request

Name: HTTP Request

Comments:

Web Server

Protocol [http]: Server Name or IP: localhost Port Number: 5000

HTTP Request

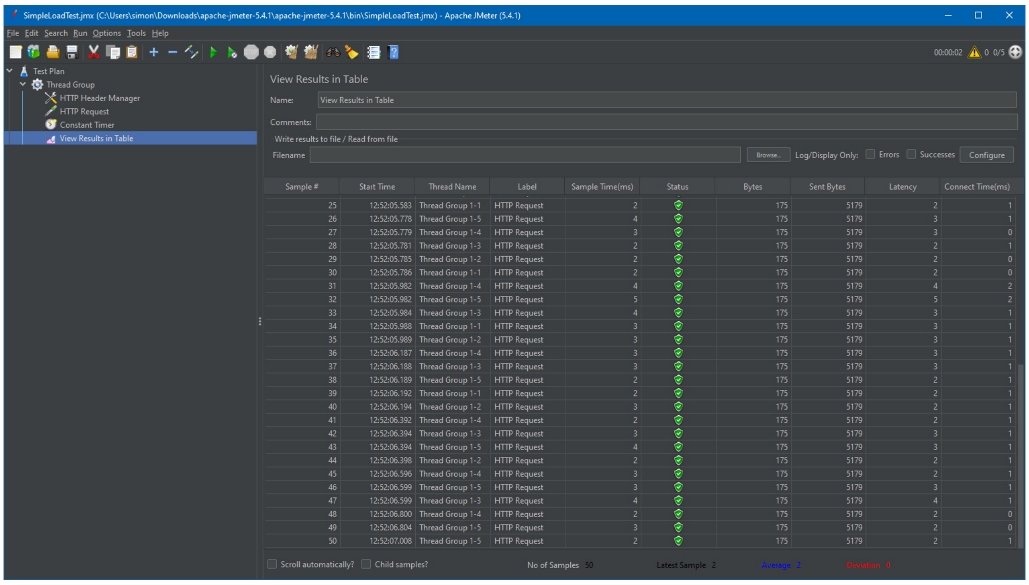
POST Path: /testPost Content encoding:

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

```
${__RandomString(5000, abc)}
```

Figure 2.4: Simple test plan in JMeter

To illustrate the usage of JMeter with a concrete example, figure 2.4 shows a test plan setup that could be used for a simple load test. The surrounding Thread Group is configured with a thread count of five and a loop count of ten, meaning that the test will simulate five parallel users and will be run ten times. First inside the group is a HTTP Header Manager setting the header ‘Content-Type‘ to “application/json”. After that, the HTTP Request element specifies the name of the server “localhost” together with the request itself, which in this case is a POST request to the path “/testPost”. Inside the body of the request is a special built-in function in JMeter that generates a random string of length 5000 using the characters “abc”. Finally, there is a Constant Timer, which puts a 200ms delay between each test loop, and a Listener of the type ‘View Results in Table‘, seen in figure 2.5, that shows the sent requests together with analytical data like the number of sent bytes or the response time of the server.



SimpleLoadTest.jmx (C:\Users\simon\Downloads\apache-jmeter-5.4.1\apache-jmeter-5.4.1\bin\SimpleLoadTest.jmx) - Apache JMeter (5.4.1)

View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

Filename

Log/Display Only: Errors Successes

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
25	12:52:05.583	Thread Group 1-1	HTTP Request	2	✓	175	5179	2	1
26	12:52:05.778	Thread Group 1-5	HTTP Request	4	✓	175	5179	3	1
27	12:52:05.779	Thread Group 1-4	HTTP Request	3	✓	175	5179	3	0
28	12:52:05.781	Thread Group 1-3	HTTP Request	2	✓	175	5179	2	1
29	12:52:05.785	Thread Group 1-2	HTTP Request	2	✓	175	5179	2	0
30	12:52:05.786	Thread Group 1-1	HTTP Request	2	✓	175	5179	2	0
31	12:52:05.982	Thread Group 1-4	HTTP Request	4	✓	175	5179	4	2
32	12:52:05.982	Thread Group 1-5	HTTP Request	5	✓	175	5179	5	2
33	12:52:05.984	Thread Group 1-3	HTTP Request	4	✓	175	5179	3	1
34	12:52:05.988	Thread Group 1-1	HTTP Request	3	✓	175	5179	3	1
35	12:52:05.989	Thread Group 1-2	HTTP Request	3	✓	175	5179	3	1
36	12:52:06.187	Thread Group 1-4	HTTP Request	3	✓	175	5179	3	1
37	12:52:06.188	Thread Group 1-3	HTTP Request	3	✓	175	5179	3	1
38	12:52:06.189	Thread Group 1-5	HTTP Request	2	✓	175	5179	2	1
39	12:52:06.192	Thread Group 1-1	HTTP Request	2	✓	175	5179	2	1
40	12:52:06.194	Thread Group 1-2	HTTP Request	3	✓	175	5179	2	1
41	12:52:06.392	Thread Group 1-4	HTTP Request	2	✓	175	5179	2	1
42	12:52:06.394	Thread Group 1-3	HTTP Request	3	✓	175	5179	3	1
43	12:52:06.394	Thread Group 1-5	HTTP Request	4	✓	175	5179	3	1
44	12:52:06.398	Thread Group 1-2	HTTP Request	2	✓	175	5179	2	1
45	12:52:06.596	Thread Group 1-4	HTTP Request	3	✓	175	5179	2	1
46	12:52:06.599	Thread Group 1-5	HTTP Request	3	✓	175	5179	3	1
47	12:52:06.599	Thread Group 1-3	HTTP Request	4	✓	175	5179	4	1
48	12:52:06.800	Thread Group 1-4	HTTP Request	2	✓	175	5179	2	0
49	12:52:06.804	Thread Group 1-5	HTTP Request	3	✓	175	5179	2	0
50	12:52:07.008	Thread Group 1-5	HTTP Request	2	✓	175	5179	2	1

Scroll automatically? Child samples? No of Samples 50 Latest Sample 2 Average 2 Deviation 0

Figure 2.5: Results of the described test

2.3.3 Usage

The use of load tests serves two main purposes for this thesis: To help with the analysis of performance metrics and to compare the effectiveness of scaling approaches.
In order to get a reliable set of system metrics that accurately represent the system performance, they need to be analyzed using real data sets generated during usage. While monitoring these values during regular production periods forms a good basic understanding, it usually does not show how the behavior changes under special circumstances like a load spike or a number of specific requests. By simulating these scenarios using load tests, the selected set of metrics can be better monitored and evaluated towards their usefulness and accuracy.
A similar situation is presented by the evaluation and comparison of scaling approaches, which is also a core goal of this thesis. As testing a novel auto-scaling implementation on production systems is risky and also limited in terms of experimentation, a custom testing environment is a more feasible approach. Here, handcrafted scenarios can be simulated using JMeter to see how the different scaling concepts react and how the system behaves accordingly.
As for the concrete choice of software, JMeter was the preferred load testing tool for a few different reasons. Firstly, a previous bachelor's thesis at the company that was concerned with creating load tests for one of the products that is also used in this thesis used JMeter, so these tests could be reused as a basis of evaluation. Additionally, JMeter is powerful but simple to use and also widespread, so many helpful resources and documentation exist to ease with its' use.
Another tool that was considered as an alternative was Gatling [GLG], which is very similar to JMeter and uses Scala scripts as a way to define tests. But due to this similarity together with its' significantly more complex usage, JMeter was the simpler and better suited choice for this use case.

3 Related Work

3.1 Existing research

Since the concept of fully automatic scaling of server resources is still rather new, the research in the field is also comparably young and innovative, with many different ideas and concepts being explored and tested. As the focus of this thesis is on a concrete practical implementation, the following summary of related work is separated into sections corresponding to the different aspects of an auto-scaling application.

3.1.1 Metrics and analysis

Scaling actions are performed once the currently operating set of resources is no longer adequate for a given workload, either because there is an overload of work, causing a delay in request processing and a degradation of service quality, or contrarily because of a lack of sufficient work, leaving units idle with nothing to process. These unfavorable states need to be detected using metrics, that can be tracked and set to trigger actions once they pass a certain threshold. Those metrics can be put in either of two categories, one looking at the current load and performance of the system, and the other looking at the current workload that needs to be processed.
The most common method for measuring system load is to look at hardware usage, usually the current CPU utilization, as it is representative of the system's current processing capacities. Therefore this metric is available in some industry auto-scaling solutions, as described in section 3.2, and also in Kubernetes as the key metric for it's Horizontal Pod Autoscaler (HPA) [K8S]. In it's basic form, the algorithm attempts to keep the relative CPU usage of each running instance at a target value by either adding or removing pods as needed. Casalicchio et al. [CP17] note that the use of relative instead of absolute CPU usage can possibly lead to violations of service level agreements, as expected response times calculated using the relative values will always be lower than the real response times. As a consequence, they propose an altered version of the scaling algorithm using absolute CPU usage leading to improved response times in testing. Klinaku et al. [KFB18] additionally note that the HPA's reliance on CPU usage as a trigger for scaling leads to a trade-off between speed and precision, as the number of pods to deploy is calculated by summing up all relative usages and dividing the result by the target usage value. Consequently, if a high target utilization is set, the algorithm will perform smaller steps, which are precise during calm periods, but can lead to slow scaling when a big derivation in load occurs. On the other hand, a low target value will lead to bigger and faster steps good for dealing with these abnormalities, but which will be struggling to keep a steady number of pods during phases with little load derivations.
The second approach to triggering scaling actions is to look at the current workload, which in most common scenarios can be measured by counting some form of requests arriving at the servers, for example the number of user requests on a website. A simple approach proposed by Assuncao

et al. [DDB09] counts all currently incoming requests and then assigns the minimum number of instances needed to process those requests within their given deadlines. As most services store their incoming tasks in queues, additional measures like queue size, wait time in the queue or overall response time can be measured. Marshall et al. [MKF10] use different combinations of these times, with regards to the time needed for instances to start up or shut down, to calculate the amount of resources needed to process the request queue as efficiently as possible. While relying purely on these higher level metrics is also not fully reliable, for example if the scaling system assigns too many new resources during rapid queue growth [KFB18], they are arguably more suited and intuitive when building an auto-scaling solution to satisfy certain service level agreements.

3.1.2 Scaling policies

The arguably most interesting aspect of auto-scaling research is finding strategies and rules on when and how to actually perform scaling actions of different kinds. Therefore many diverse approaches have been proposed and also evaluated in the research field.
One already mentioned approach, the Horizontal Pod Autoscaler in Kubernetes [K8S], is comparably simple but representative in its methodology for other scaling policies. First of, a monitoring system checks the current relative CPU usages of the pods in small regular intervals, usually around thirty seconds, and combines this data with a given target CPU usage to estimate the desired number of running pods. If this amount deviates from the currently active number of pods, the necessary scaling actions are performed, after which the system goes in a short cool down period where no other actions are performed to avoid oscillation effects from rapid contrary scaling operations. While the scaling results are not always optimal [CP17], the system represents a common implementation of the MAPE Control Loop [JLNY04]: The HPA monitors (M) the CPU usages, analyzes (A) them in comparison to the target value, plans (P) the required number of pods to meet the goal and finally executes (E) the necessary scaling operations. This loop structure is the basis of most auto-scaling systems and is sometimes expanded to a MAPE-K Control loop [MKS07] to emphasize the role of human knowledge (K) and expertise needed in conjunction to operate a successful auto-scaling system.
This improved loop is for example used by Klinaku et al. [KFB18], where they calculate their required number of containers by dividing the current workload intensity by the capacity of the containers, i.e. the amount of tasks a single container can handle. This capacity is evaluated by experts of the running application and its context to provide an accurate estimation. In order to deal with short load spikes and to give the system time to start the new containers, they assign a relative number of additional spare containers as assistance. Overall, their auto-scaling solution has shown to over provision most of the time, but with a very high accuracy i.e. very few idle containers.
Marshall et al. [MKF10] propose other policies that are based on a monitored job queuing system, which puts jobs that can currently not be completed due to a lack of resources into a waiting queue, which then triggers scaling actions based on properties like wait time and start-up/shut-down times. Their simplest approach starts one additional instance when a job is queued, indicating that more resources are required, and terminates all idle instances once the queue is emptied again. The "steady stream" policy keeps additional instances running to handle shorter load spikes and only starts additional instances once the total wait time of jobs exceeds a factor of the time needed to start these additional instances, while likewise being conservative in scaling down as well by taking shut down times into consideration. The third approach works in a similar way by adding bursts of additional instances as they are needed when load spikes arrive. Evaluation of these policies

show that the steady stream approach takes time to adjust itself to the workload but keeps a steady number of instances running once it has tuned itself, while the other two approaches tend to closely match the current workload, causing significant fluctuations and oscillation during irregular request spikes. They also note that over provisioning with additional instances in advance and performing scaling operations in parallel can significantly reduce delays caused by the auto-scaling system.
A more generalized evaluation of auto-scaling strategies by Netto et al. [NCCA14] also discusses properties of using differently sized and calculated step sizes for performing scaling operations. They conclude that using adaptive step sizes, which add or remove a dynamic number of instances per scaling, are better suited for irregular workloads, while using fixed numbers of instances is simpler and well sufficient for regular workloads, as they can lead to over-/under provisioning during short load spikes. They furthermore emphasize the importance of defining proper thresholds for system performance metrics, preferably defined manually by qualified personnel.
The research of Mao et al. [MH11] also points to some general good practices that can improve the performance and efficiency of a system reliant on auto-scaling. Their results show that in many cases, it is actually more efficient to not perform any scaling operations and instead to queue jobs and execute them on the already running machines, as the start-up times of additional machines often exceeds the wait times of the jobs themselves, which is a case that has been previously explored by Marshall et al. [MKF10]. Additionally, bundling jobs of the same type on machines with hardware optimized for their use case allows for more precise and optimized scaling actions [MH11].

3.1.3 Evaluation methods

To properly evaluate and optimize an auto-scaling system, an adequate measurement of the resulting system performance and user experience is required. One commonly used performance metric for this case are the response times of user requests, as applied for example by Assuncao et al. [DDB09]. Using different timings like execution or submission time in combination with the available resources needed for completing the task, they calculate an average weighted response time over a set of requests to get an impression of the current service quality on the user's end. A similar goal is pursued by Marshall et al. [MKF10], who aim to minimize the job turnaround time while simultaneously minimizing oscillation effects and avoiding idle machine time.
Another common metric to represent the quality of an auto-scaling system is some form of matching function between the currently dedicated amount of resources and the optimal number that would be needed to adequately process the current workload. One realization of this function is the auto-scaling demand index defined by Netto et al. [NCCA14], that tracks the difference between the current resource utilization level and the one targeted by the scaling system, leading to both penalties for expensive over provisioning and a decline in service quality for under provisioning.
Herbst et al. [HKR13] go even further by extensively defining the property "elasticity", referring to the ability of a system to adapt itself by autonomously matching its resources to the current demand as closely as possible. The elasticity of a system consists of two different dimensions, with precision simply showing the difference between the allocated resources and the actual workload, and speed being manifested by the time it takes for the system to move from an over- or under provisioned state to an optimal one. On the mathematical side, they define these properties using sums and averages of mismatched resources and time spent in sub-optimal states, which are tracked over time and then used to calculate precision and elasticity values for scaling resources up and down respectively. Furthermore, they stress the importance of equal demand curves and performance goals during comparison of different auto-scaling systems to ensure equal stress and comparable

benchmark results.
An effort to formalize the properties of an elastic system done by Bersani et al. [BBD+14] adds additional metrics fitted for defining but also evaluating the performance of auto-scaling methods. Starting off with measures concerning the reaction time of the scaling and the sensitivity with load measurements to trigger actions, denoted as "eagerness", the definitions go further with the term "plasticity", referring to the system's ability to return to a minimal configuration after allocating additional resources, with a lower value referring to a preferable higher speed. Apart from other familiar properties like oscillation and cool downs, they also calculate two measurements representing quality of service, with "degradation" referring to the difference in performance during scaling operations and the "actuation delay" measuring the time the system needs for additional resources to be ready after scaling up and being fully stopped after scaling down.

3.2 Current industry solutions

Apart from theoretical auto-scaling concepts developed in the research field, there are of course already many production solutions running, mostly in the cloud hosting area. The three subjects here will be Amazon's EC2 [EC2] and Microsoft's Azure [WAZ], which are two large and widely used cloud service providers today, together with Netflix's Scryer [NFX], an in-house scaling solution by the company that focuses on predictive scaling methods.

3.2.1 Amazon EC2

The core concept of EC2's scaling implementation are scaling groups, each containing a variable number of application instances. A scaler responsible for this group then ensures that a desired capacity of instances is running, which is constrained by a minimum and maximum number of instances per group. These instances can be of different types and hardware configurations depending on their use case, which allows further fine tuning to performance requirements even without scaling. To allow the system to automatically start these additional instances, certain launch templates and configurations are defined by the user, describing the application and it's setup for each instance.
The scaling behavior itself is controlled by one of a select number of available scaling policies. The most common policy is to do manual scaling, where the user specifies and manually updates a desired number of instances within the given bounds, which can be simplified further by disabling scaling all together and just maintaining a fixed number of instances at all times. More advanced policies allow the definition of scaling schedules, where scaling operations are tied to certain times and dates, or more advanced scaling rules based on user-defined thresholds on system performance metrics like CPU or memory usage. Finally, Amazon also offers a predictive scaling approach, which uses collected system metrics to predict future workloads and performs scaling actions in advance.

3.2.2 Microsoft Azure

Azure provides an extensive rule-based approach for auto-scaling cloud resources within the available capacities. First of, a scaling profile is defined by the user, specifying the upper and lower scaling bounds and when scaling operations are to be executed, which can either be set to always or be limited by a given recurring or fixed date. Secondly, scaling rules are defined using system metrics for a given resource. The rule checks if a certain metric is above or below a certain threshold, optionally aggregated over a time interval, and then adds or removes instances after which it goes into a cool down phase to prevent oscillation effects. Apart from system metrics, scaling operations can also be triggered by a message queue or another custom Azure resource.

3.2.3 Netflix Scryer

Finally, while Netflix's Scryer is not available for public use, it does provide an interesting practical implementation of predictive scaling techniques. Their motivation for Scryer arose from their dissatisfaction with reactive auto-scaling behavior during high load spikes, as scaling up too late resulted in a critical delay caused by long start-up times for the new instances and excessive down-scaling during outages left the system under provisioned once traffic spiked again after the outage. Furthermore, Netflix has a very predictable load pattern thanks to its use case, which arguably makes predictive scaling even more appropriate than reactive solutions.
The predictions of the system are created by applying a linear regression algorithm, together with Fast Fourier Transformations for smoothing, to select points from their user request data, with more focus being put on newer data to keep the predictions up-to-date. They specifically use request volumes instead of system load averages, as the latter is influenced by code changes and even the scaling itself. From these results, an auto-scaling plan is built and executed, with minor alterations for special events like holidays.

4 Case study design

4.1 Research goals

From the three main research questions proposed in section 1.3, the focus for the case study lies primarily on the first two, with the main goal of getting a clear understanding about the requirements that will shape the scaler and the metrics than could prove useful for tuning it.
When going into the requirements for a scaling solution in this context, it is important to consider the different stakeholders and how they will be affected by this new system.
The first and most important group of these are the customers whose experience using the application should not be impacted negatively by the auto-scaler. To accomplish this, one goal is to find out more about their behavior and interaction with the system, e.g. when and how much they use it, how important a responsive UI is compared to quick automated request processing or how tolerable they are towards slow performance and outages.
The second big group of stakeholders are the developers and engineers at the company, that need to be able to efficiently use the system and work with it on a daily basis. Since there are many different teams and applications though, it needs to be determined what the common goals and features expected of an auto-scaling solution are and how it can be used in their products' context. On a technical level, this also means getting a clear understanding of the possibly differing architectures of the applications, where possible bottlenecks lie in terms of scaling, and how an auto-scaler could best be applied to their case.
Moving from requirements onto the second goal, the metrics that can be used for the scaling system are another point of interest. Depending on the applications specific hardware requirements and internal structure, different points of measurement are needed to adequately determine the system's current performance and, in the future, necessary scaling operations to keep the system running at a stable level. Additionally, finding out how these properties are currently measured, what rules and threshold values are present and finally how that data could be fed into an auto-scaler are also interesting points.

4.2 Subject selection

Since tech companies are often comprised of a diverse set of people and roles, finding the right subjects was an important decision in order to create a founded and reasonable base of requirements. Due to the technical nature of the questions and topic at hand, the group of people to focus on were mostly architects and engineers familiar with the systems and scaling mechanisms. These can be split further into two groups:
Because the applications and their behavior are going to be the determining factor for how the auto-scaling will work, the developers and architects responsible for the products themselves

comprised the first group of subjects. They are the ones with the deepest understanding of the architectural properties, and also the ones with the most important set of requirements for the auto-scaler, both from their own technical perspective but also from the customers. Getting some input from support also gave further insight into the customers' experience and possible issues. Secondly, to find out more about how the systems are actually deployed and operated, the second group consisted of system admins and operators whose main focus is ensuring a smooth operation of all production systems hosted at the company. As monitoring the applications is an integral part of this, they know how to most accurately determine system performance, when there is a need to intervene and how the latter can be done in an efficient and non-disruptive manner.

4.3 Data collection procedures

Since the number of subjects was small and it was important to get detailed responses in regards to certain topics, the case study was performed using semi-structured interviews. To collect the information needed, a catalog of around thirty questions was created based on the research goals of this thesis but also more general and open questions, for example in regards to their working background and some in-depth information about specific architectures, to get a broad understanding of the subjects and applications at hand. The interviews were scheduled to be an hour long, to give enough time to go through all questions and to allow opportunities for conversation and additional questions.

5 Data analysis

5.1 General

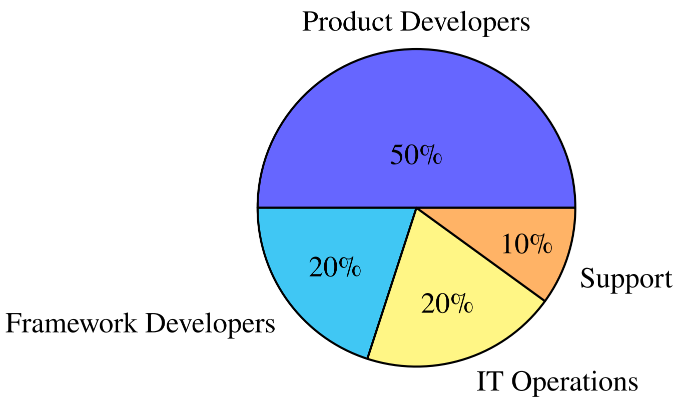


Figure 5.1: Distribution of interview subjects

The interviews took place over seven different sessions in the span of two weeks at the beginning of 2021. In each session, one or two people from a specific team were interviewed, with ten interviewees overall. Four sessions were dedicated to four different main products, one session for the team working on framework technology, another session with the IT operations team and finally one session with the support team. The subjects consisted mostly of developers, with five software architects and three regular programmers, who are all long-time employees, with an average of 20 years at the company.

5.2 Requirements

Requirement description	**Number of mentions**
Quick response times for automated requests and GUI interaction	5
Manual intervention, exceptions for specific times/customers	4
Self-adaption of scaling rules	3
Ability to scale in/down	2
Configuration via SLAs e.g. guaranteed response times	2

Table 5.1: Most common requirements

5.2.1 Customers

A large portion of AEB customers are big companies that mostly rely on their automated systems, with around 90% automation for their Customs products, while the fewer smaller customers mostly prefer using the UI of their products, with around 75% of their Compliance product users preferring the UI. Due to the recent Brexit, there has been a big increase of smaller customers, especially for their Customs products, resulting in more UI users there as well.
The recent events have also lead to an general influx in system usage, resulting in an overall moderate to high usage, with Compliance running at around 85%. These usage patterns closely follow a day-night cycle and most derivations or peaks are well predictable beforehand.
Since the consequences for system outages can be pretty severe, causing major logistical problems or heavy fines, the amount of outages in recent years has been low, with the interviewees only being able to recall four major events, either caused by performance issues or a combination of bad user input and software errors. Apart from long response times for specific Compliance requests, customer feedback related to performance has been positive throughout.

5.2.2 Architecture

The architecture is entirely service-oriented, with different modules that communicate with messages and that can be run separately. Apart from the core modules of the products, there are also external components like label printers or customs systems, which are often limiting performance factors due to slow printing or request limits from customs.
Another bottleneck from the product developers' perspective is the database, as it limits scaling possibilities and is slow in processing long requests, while the framework team instead assures that the database is already optimized well and only appears slow due to incorrect usage.
Because the current architecture wasn't built with containerization or auto-scaling in mind, many aspects do not carry over well to a cloud environment. While the ability to run applications in separate modules and to easily transfer sessions between servers are beneficial, one major limitation is that the architecture heavily relies on states, meaning that sessions and processes are linked to specific machines, making it difficult to dynamically add and remove them without causing issues. Additionally, the number of parallel UI sessions is very limited and long start-up times impact the efficiency of scaling operations as well.

5.2.3 Specific requirements

While four interviewees would rate the current scaling solution as ‘good‘ and three as ‘acceptable‘, everyone agrees that automatic scaling would be beneficial in many aspects.
The most commonly requested features for an auto-scaler were the ability to intervene manually and to set exceptions for specific times or customers, options to scale in/down, improved system monitoring options and a more modernized approach to system operation overall. Regarding the latter, five interviewees stated that they have little to no experience with Kubernetes and Prometheus, while IT operations and framework developers are more confident with those technologies.
In terms on configuration options for the auto-scaling, most developers would like to be able to input specific service-level agreements, e.g. a guaranteed response time for Carrier products, which the

scaler then tries to meet by estimating the amount of resources needed, for example using queuing theory. Additionally, a fully adaptive scaling system that learns and adapts its' thresholds and rules dynamically was considered as well.

5.3 Performance estimation

Metric description	Number of mentions
Absolute CPU usage (in %)	7
Amount of used memory/heap space	6
Response/message processing times	3
Number of queued jobs/messages	3

Table 5.2: Most commonly used performance metrics

5.3.1 Metrics

The most important hardware metrics for all teams are CPU usage and RAM allocation, with the latter being the resource that the framework uses the most. IT operations additionally looks at hard drive usage and, on an application level, continually monitors reserved heap space of the systems together with the number of running background tasks and their error rates. In software measures, the product developers look at response times and queues, e.g. the number of queued jobs or the amount of unprocessed customs messages.
Additionally, support also measures UI response time to get a closer insight into the user experience.

5.3.2 Monitoring

Product teams do not regularly monitor all their system performance stats, but only continually look at certain metrics critical to their application, e.g. CPU usage for Carrier or response times for Compliance products. The monitoring is executed using self-made scripts and alerts, although only few threshold values are specifically set, e.g. a 25% CPU usage per node for Carrier or a 15 second response time limit for Compliance products, and instead the teams become active when they feel that the monitored values are reaching problematic levels.
IT operations on the other hand performs much more extensive and regular system monitoring, with in-depth performance analyses on a weekly basis. They determine their threshold values for alerts dynamically over time, with a big focus on proactive monitoring to detect possible performance problems early. They note though that exhaustive manual monitoring is not feasible for the amount of machines running, except for critical time periods like the Brexit phase.

5.4 Current scaling solution

Right now, all scaling is done manually by IT operations, usually based on a request by the product developers or support. The scaling operations are done conservatively and permanently, so no down- or in-scaling is performed. Since the framework relies mostly on RAM, vertical scaling is for the most part accomplished by allocating more memory to a VM, while horizontal scaling is done by adding additional VMs. If a specific VM is causing performance issues, it can also be isolated to prevent it from clogging up resources needed for other VMs.
One major part of avoiding scaling all together is a proper estimation of needed resources before a new system is installed for a specific customer. Because performance tests tend to become outdated rather quickly, these estimations are mostly based on experience by the developers and IT operators.

5.5 Discussion

5.5.1 Requirements

From the customers' side, there are two main performance requirements that the scaling solution should be able to reach: The bigger customers need quick processing for a large number of automated requests, while smaller customers put a strong emphasis on few but responsive UI sessions. Additionally, avoiding outages and poor performance is critical to not cause problems in logistical processes, so the auto-scaling system should lean more towards over-provisioning to ensure that enough resources are always available. Additionally, scaling in and down would be preferable as well, but could be difficult to implement due to the applications' stateful architectures. Because the usage patterns are regular and closely related to the day-night cycle, a predictive scaling approach is a viable option, together with the requested ability to add product- or time-specific exceptions to intervene manually.
Since most developers have little to no experience right now with Kubernetes, the configuration options of the auto-scaler should be simple and mostly focused on the ability to input known service-level objectives like response times and have the system do the rest, possibly adapting itself over time. Extensive configuration beforehand to tailor the auto-scaling to estimated requirements for a given customer should also be possible to reduce the need for scaling overall.
When looking at how to do the scaling itself, matching the current manual scaling operations closely is a good starting point, with vertical scaling focused on RAM and horizontal scaling in the form of additional VMs.
The architecture of the applications could be a major limiting factor for the scaling system, since it was not built with those concepts in mind and relies heavily on stateful session management while also boasting considerably long start-up times. Still, the service-oriented architecture with the ability to run modules separately could prove beneficial.

5.5.2 Metrics

In terms of metrics, the most important hardware based metrics are CPU and RAM usage, with the latter being a good indicator of how much work the framework is currently doing. Additionally, monitoring hard drive space could also be a performance indicator, although it is not as directly related to the current system usage as the other two metrics.
On the application level, the key measurement for performance and user experience is the response time, both when processing automated requests and when interacting with the UI, and is often fixed by service-level agreements providing limits that the auto-scaling can use as thresholds. The current length of job and message queues, together with the overall number of running jobs and their error rates, are also good indicators which can be used to estimate response times indirectly.
Finally, the current monitoring options need to be expanded and adapted to allow for continual monitoring of these specific metrics and to perform predictions which can then be used by the auto-scaler to schedule needed operations. The actual trigger values can either be defined manually by the user, e.g. via a guaranteed response time, or be adapted dynamically through a learning approach.

5.5.3 Limitations

As with all interviews, the subjects only make up a hopefully representative subgroup of the overall target users, so certain validity risks to the collected data are present.
First off, while the subjects were notified with an overview over the interview topics a few weeks beforehand, very few of them have had actual hands on experience with Kubernetes and the concepts of auto-scaling in general. This means that some of their answers and ideas are only estimations or guesses about how this scaling solution could work with their systems at hand.
This limitation is also linked to an admitted disconnect of knowledge between the product and IT operations teams, since the developers are usually not monitoring their systems regularly and thinking about scaling, while the operations staff does exactly that but has a much lighter understanding of the architecture and software actually running inside their VMs. Some of the interviewees have stated that they hope the new scaling solution will help both sides get a more common understanding and interaction with the topic of scaling their products in the future.
Another important limitation that arises from the open structure of the interviews is that there is a high variety in the responses. For the sake of aggregation and finding consensus in the data, responses which are different in formulation but share the same core idea are put into the same equivalence class. While most answers could be clearly interpreted as equivalent, there is a margin of error that arises from the possibility of misinterpreting certain answers.

6 Execution

6.1 Choice of application

One of the core questions for the implementation of the auto-scaler was choosing the right application to use as a subject for scaling. While the range of production applications being developed and hosted at AEB is diverse, the choice ultimately came down to their “XNSG Test Product”, which is not an actual application being sold to customers, but an empty shell of their framework for testing new engine features. There are two main reasons for this choice:
Firstly, installing a regular production application not designed for containerization onto a Kubernetes cluster is complex and requires a substantial amount of setup time to create an environment that allows the application to run and enable all functionalities it provides. This conclusion came from an attempt to set up an installation of their Customs product in the cluster, which ultimately failed due to core engine functionality behaving differently on Kubernetes pods compared to the old approach using regular Windows VMs. Solving these issues in the time and resource scope of this thesis was simply not feasible, especially when taking the second argument into account.
To effectively test a production application as complex as their Customs solution, a decent amount of mocking is required to actually fully process a realistic user request. As an example, this includes the setup of user accounts, fully mocked customs-declarations and stubs for all needed endpoints the system communicates with. Even if these testing capabilities were created, there would be no guarantee that they would put load on specific components of the system that the auto-scaler is configured to react to, making efficient testing and evaluation difficult. Since the chosen application is already designed as a test product though, it provides a set of test functionalities substantial enough to put enough directed stress on the system to trigger the auto-scaler. For this use case, an existing test method implemented as a call to the applications REST API was expanded to allow for different ways of putting load on the system, including a POST payload of variable size, a set processing time with or without CPU load and a desired amount of JVM Heap space the system should reserve for processing the request.

6.2 Architecture

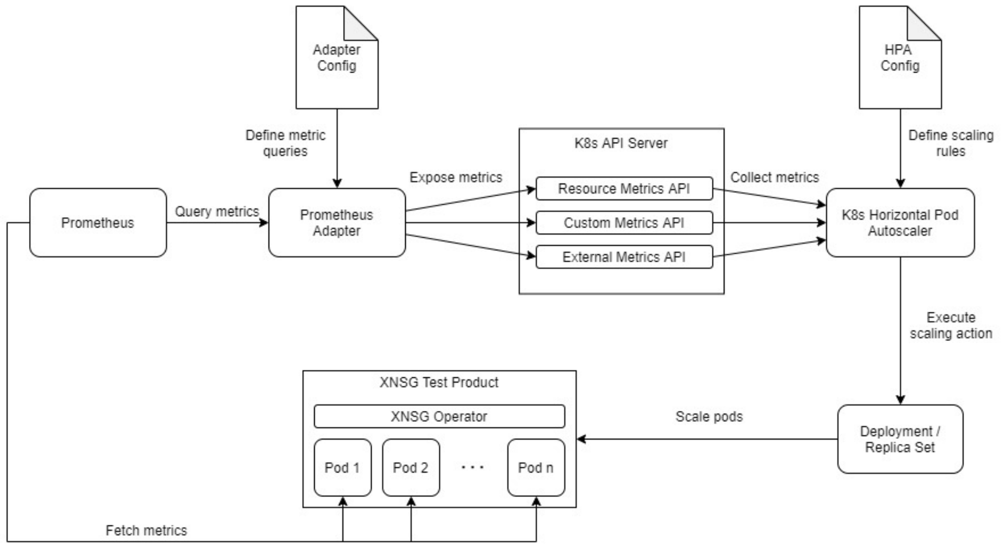


Figure 6.1: Architectural overview of the scaling implementation

6.2.1 Application setup

The XNSG Test Product is hosted in its own namespace, consisting of a variable number of regular pods, hosting the application instances, and a special operator pod, responsible for administrative tasks like performing database and software updates or restarting crashed pods. To achieve this, the XNSG Operator manipulates the deployment resource, which is interpreted by Kubernetes e.g. for scaling the number of pods and deploying revisions.

6.2.2 Implementation of metrics

Each pod and the application running inside it regularly report a list of metrics to a Prometheus instance. These metrics include raw hardware data collected by K8s, like CPU usage or total RAM usage, more specific software metrics, like the reserved and used JVM Heap space, to concrete metrics collected by the XNSG application itself and reported through its statistics framework, like the response times of specific REST API calls. The collection of metrics is stored in the Prometheus instance, where it can be accessed via queries by other components, like Grafana or the Prometheus Adapter [PMA].
The “Prometheus Adapter for Kubernetes Metrics APIs” [PMA] is a special component that acts as a connection between Prometheus and the Kubernetes Metrics APIs by querying Prometheus for specific metrics and then exposing the results via the corresponding Metrics API. The queries are defined in a configuration file using PromQL and YAML, together with the targeted API points and resources. In this use case, the adapter has been set up to collect pod CPU and memory usage and to expose them through the Resource Metrics API, therefore replacing a regular Metrics server.

Additionally, there is a set of custom metrics, containing both raw values from Prometheus but also more aggregated ones using PromQL queries, like a 99 percentile of request response times and insights into current JVM Heap space usage.

6.2.3 Implementation of scaling

Scaling is performed by Kubernetes' own Horizontal Pod Autoscaler, deployed for the given namespace. The HPA communicates with the Metrics APIs and collects the metrics defined in its configuration. Thanks to the communication over the Metrics APIs, the sources of the metrics are completely invisible to the HPA and both the HPA and the Prometheus Adapter are independent of each other. If values received by the HPA cross its' set thresholds, necessary scaling operations are triggered according to the minimum and maximum amounts of pods and the configured scaling behavior. In practice, the HPA then modifies the corresponding deployment, which then scales to the requested number of pods.

6.3 Scaling configuration

6.3.1 Prometheus Adapter configuration

The configuration for the Prometheus Adapter is based on the default values provided with the adapters' Helm chart [AHM]. To connect the adapter to the existing Prometheus installation, the adapter needs the URL and port of the corresponding endpoint. In the rules section, the following queries are defined:
First of, the default ruleset is enabled, meaning the adapter will look for and collect all already existing metrics defined in Prometheus, which also includes all metrics provided by the AEB applications. Since these do not come from a Kubernetes object itself but instead from the application running inside it, the more specific queries are defined in the external metrics section. Here, an aggregation is performed on the metric 'xnsg_bf_ms_bucket' to get a 99% percentile of all response times of recent requests. Additionally, a sum over the rate of 'jvm_used_memory_bytes' over the last three minutes is collected to get information about significant short term changes in JVM memory usage.
Additionally to the default and external metrics, both resource metrics, CPU and memory usage, provided with the Helm charts' default values are enabled, serving as a replacement for their counterparts usually provided by the Kubernetes metrics server.

6.3.2 HPA configuration

The Horizontal Pod Autoscaler is configured for the 'xnsgtest-4-0-master' deployment and given a range of replicas from two to four pods. Since proper custom scaling behavior is only supported starting with Kubernetes 1.18 [CFG], the only other modifications are related to the three scaling metrics. The choice of metrics originates from the results of the interviews and also the given technical possibilities, while the thresholds were evaluated through experimental benchmarks. From the listed metrics, only one is enabled at a time.

Resource type	**Metric name**	**Target value**
CPU resource	Average utilization	160% (i.e. 80% per CPU)
External resource	JVM Used Memory Bytes (Rate)	+3MB
External resource	XNSG BF Response time (99% Quantile)	1s

Table 6.1: Configured scaling rules for the HPA

All of the chosen metrics were mentioned in the interviews and thus chosen for the scaler. The first metric is the average over the absolute CPU utilization per pod, which goes above 100% because Kubernetes assigns two virtual CPUs to each pod, meaning the maximum possible CPU usage per pod is 200%. For the memory metric, the memory usage of the JVM was chosen, since this more closely relates to the actual RAM usage of the application compared to just the pod's overall memory usage. Since the overall variation is very low, a rate function is used to better detect the short term changes in memory usage and to be more robust overall for the scaler, since pre-defined values like "max. 800MB RAM usage" are difficult to maintain and tend to change through updates to the implementation. The final metric is the 99% quantile of the response times of all requests to the HTTP interface of the application in the last three minutes. This form for the metric using quantiles was chosen since it intuitively resembles typical service-level agreements and is more robust to outliers compared to for example just taking the average over all recent response times.

7 Evaluation design

7.1 Evaluation objectives

The core goal of the evaluation is to find out which of the chosen scaling approaches is likely to produce the best results when applied in practice. To measure this level of quality, multiple criteria need to be considered.
Firstly, as this scaler has a specific practical use case, it is important that it performs well in a multitude of different scenarios that closely resemble it's expected production environment. The shapes of these scenarios can be deduced from a combination of common load patterns and the requirements mentioned in the interviews in section 5, which give an idea of the typical load distribution on the scaled system during different times of the day. Since a scaling system is designed to keep the system operational during critical high load sections, the scenarios also need to be powerful enough to put the required amount of stress on the system to trigger a corresponding reaction by the scaler.
One major part of an accurate scaling solution is a precise understanding of the current system load. Here, the metrics used for a scaling approach need to be examined to see how they behave under different influences from the outside, but also how they possibly change as a reaction to actions by the scaler itself. This accuracy can simply be measured by comparing the patterns of the given load scenario to the one of the collected metric.
Since a highly accurate metric might not automatically lead to a better scaling performance though, it is also essential to observe the end result of the entire scaling process, which is the actual experience of the user interacting with the system. Because real user satisfaction is dependent on many different factors and thus difficult to measure, the target here will be to fulfill given service-level objectives, as they are the deciding criteria in practical scenarios with real world production applications.

7.2 Experiment design

7.2.1 Scenarios

The two chosen scenarios for this test are based on the ones used in the evaluation of Klinaku et al. [KFB18] for their scaling solution. They were chosen because they reflect two commonly occurring load patterns, which can also be regularly observed at AEB according to the usage patterns reported by both engineers and IT admins during the conducted interviews. In order to implement them more easily using JMeter, the "Ultimate Thread Group" from the Custom Thread Groups plugin for JMeter [CTG] was used.

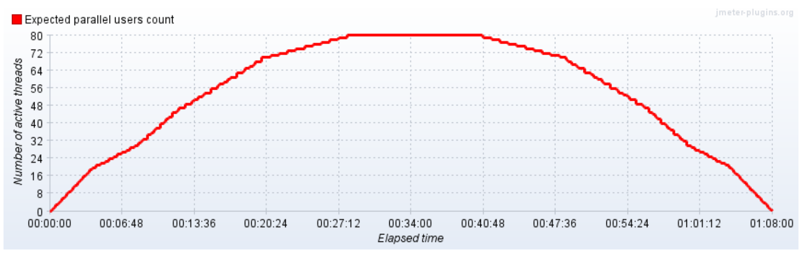


Figure 7.1: Load distribution for the Increase-Decrease scenario

The "Increase-Decrease" scenario is a simple wave form, with a slow increase in request volumes until a high point is reached and held for a while, after which the load decreases again with a similar speed. This pattern closely resembles a typical day-night cycle of load distribution, with rising load until noon and then a drop afterwards towards the evening, leaving little to no load for night time. The applications at AEB follow a very similar usage pattern throughout a day-night cycle, which makes this scenario the most representative for a real world load distribution and thus a vital testing scenario for the scaler. Naturally, the expected scaling behavior is a scale-out during the increase in load, followed by a scale-in after the load peak has passed. To give the scaled-out resources enough time to redistribute the high load, the high load time is purposefully stretched out over a longer time.

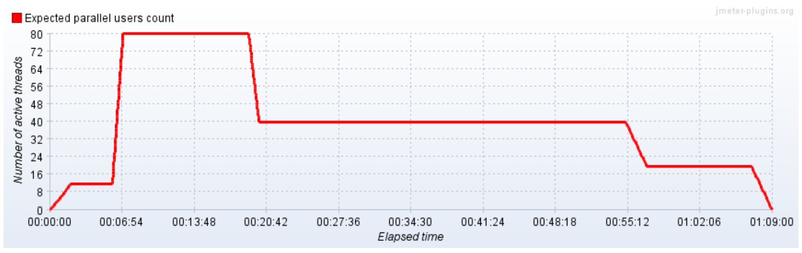


Figure 7.2: Load distribution for the Spike scenario

The "Spike" scenario on the other hand presents a more extreme situation, starting with a short period of low load followed by a huge spike in requests, which slowly drops down again over an extended period of medium and finally low load. While this scenario might not occur often in practice, it is still vital that the scaler is able to deal with it and reacts in a sensible manner. One possible origin for a similar load spike is a "retry storm" after a service outage, as described in Netflix' article [NFX], after which an initial huge flood of requests follows, which takes some time to process until regular usage levels are reached again. To preserve service quality during this time, the initial scale-out reaction by the scaler should come as quickly as possible, followed by a corresponding scale-in once the spike has been resolved.

7.2.2 Test environment

Target for the tests is a Kubernetes deployment as described in section 6.2.1, physically spread across four Kubernetes nodes. The examined deployment runs in its own namespace together with two additional pods, hosting the Prometheus Adapter and the XNSG Operator respectively. At it's minimal configuration, the deployment consists of two application pods, with each hosting one instance of the XNSG Test application described in section 6.1. The Prometheus Adapter fetches performance metrics from all pods in the namespace, with application-specific metrics being naturally only provided by the application pods. The Horizontal Pod Autoscaler is configured as described in section 6.3.2 and thus only sees the deployment i.e. the application pods, with the other pods like the adapter and the XNSG Operator being ignored for scaling.

7.2.3 Test execution

Before each test, the deployment is scaled down to its minimum of two pods and the system load is kept at a minimum. For each test run, a new HPA instance is applied, with one of the rules described in section 6.3.2 enabled. Once the scaler is running, one of the testing scenarios is started using JMeter. Each scenario maxes out at 80 parallel users and runs over a span of 60 minutes. Each user repeatedly sends HTTP requests to the testing API of the application, consisting of a randomized 256 character string and a set processing time of 500ms, during which load is put on the CPU of the application hardware. After each scenario has run through, the system is left idle until the scaler returns to the minimal number of pods again. To increase result confidence, each combination of scenario and scaling configuration is run three times, taking an average over all three test runs for the evaluation. The result data is extracted from JMeter and Grafana, with Grafana providing the majority of metric values and JMeter listing the raw response times for each request.

7.2.4 Evaluation criteria

For evaluating the quality of the metrics themselves, their correlation with the actual system load is the most important measurement. This can be measured intuitively by plotting the two data sets together and comparing their shapes, and more precisely by calculating their Pearson correlation coefficient. If the two highly correlate, the chosen system metric accurately represents the actual system load.
The quality of the scaler itself is evaluated both from an inside and from a user perspective. Eagerness and Plasticity [BBD+14] both provide information about the reactivity of the scaler, with eagerness defining the time it takes for the scaler to react after its' metric thresholds have been breached, and plasticity measuring the time it takes for the scaler to scale down again after load subsides. For the user perspective, the mean time to repair after a failure, i.e. a breach of a service-level objective, is relevant. The defining objective for this will be a maximum 1 second response time for 90% of requests. Since the different scaling approaches might react before a breach even occurs, depending on the metric and its threshold, this time might also be negative, which can be interpreted as a "prevention" time.

8 Evaluation and Discussion

8.1 Data analysis

8.1.1 CPU-based scaling

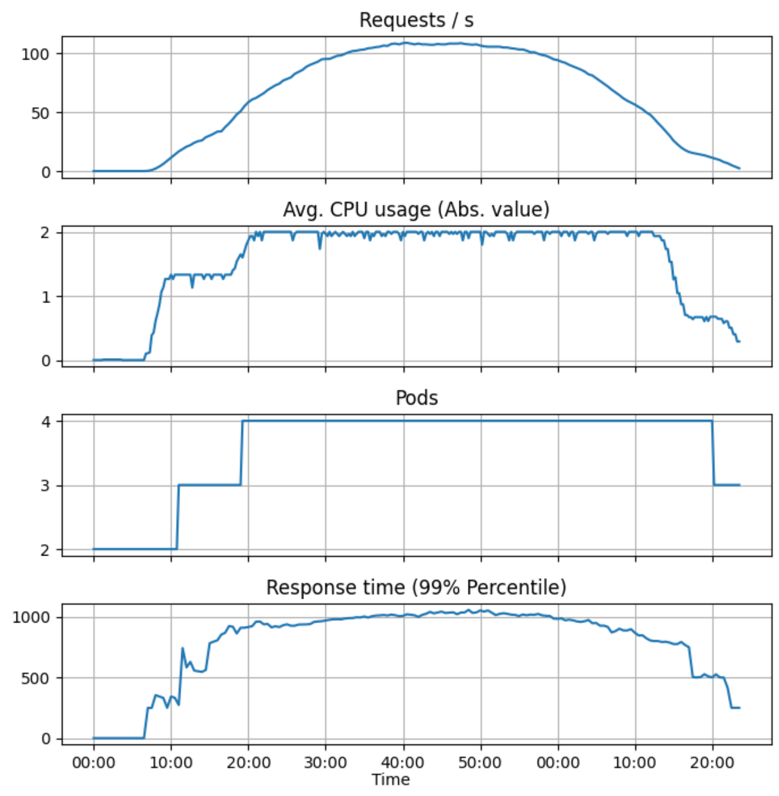


Figure 8.1: Results for Increase-Decrease scenario (CPU-based scaling)

Starting with the CPU-based scaling approach, figure 8.1 shows the average CPU utilization as an absolute value between 0 and 2 virtual CPU units, with the HPA configured to react at a usage of 1.6. Although the CPU usage it mostly maxed out when load is present, which is linked to the way the test method uses the CPU, the Pearson correlation coefficient amounts to $r = 0.87$, signaling a strong correlation and thus a good performance metric accuracy.
The reaction time of the scaler is rather short, with an Eagerness of 35 seconds and a pod start up time of 1 minute and 36 seconds. Note the long delay between the start of the third and fourth pod, indicating a more conservative approach by the scaler, likely caused by the initially low CPU

usage of the third pod, driving down the overall average CPU utilization. After all requests have been processed, the CPU utilization drops quickly, resulting in a Plasticity of 3 minutes and 15 seconds for the scaler. User experience differs greatly for this scaling approach, as the initial spike in response times is prevented by the scalers early reaction. As a consequence though, multiple smaller failure periods occur towards the load peak, resulting in a summed up average failure time of 17 minutes and 40 seconds.

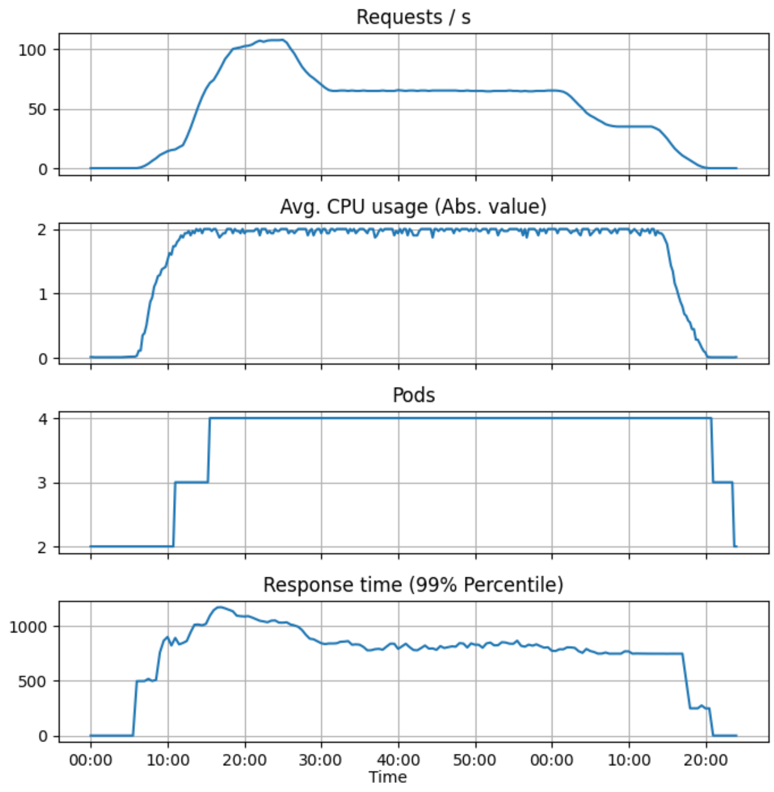


Figure 8.2: Results for Spike scenario (CPU-based scaling)

With a Pearson correlation coefficient of only $r = 0.78$, the accuracy of CPU utilization as a performance measurement drops slightly in the Spike scenario. While this fact likely arises from the high CPU usage of the testing method, it demonstrates a risk when relying on CPU utilization as a sole scaling threshold.
The reaction of the scaler varies from the one seen in the Increase-Decrease scenario, with a higher Eagerness value of 50 seconds and a pod start-up time of 1 minute and 36 seconds but a notably smaller gap between the start of the third and the fourth pod. So while not drastic, the reaction of the scaler is slightly faster for the Spike scenario. Scaling down occurs in a similarly rapid fashion, with a Plasticity of 3 minutes and 30 seconds. Looking at the response times gives a mean time to repair of 11 minutes and 40 seconds, again occurring at the load peak, only slightly affected by the additional pods.

8.1.2 Memory-based scaling

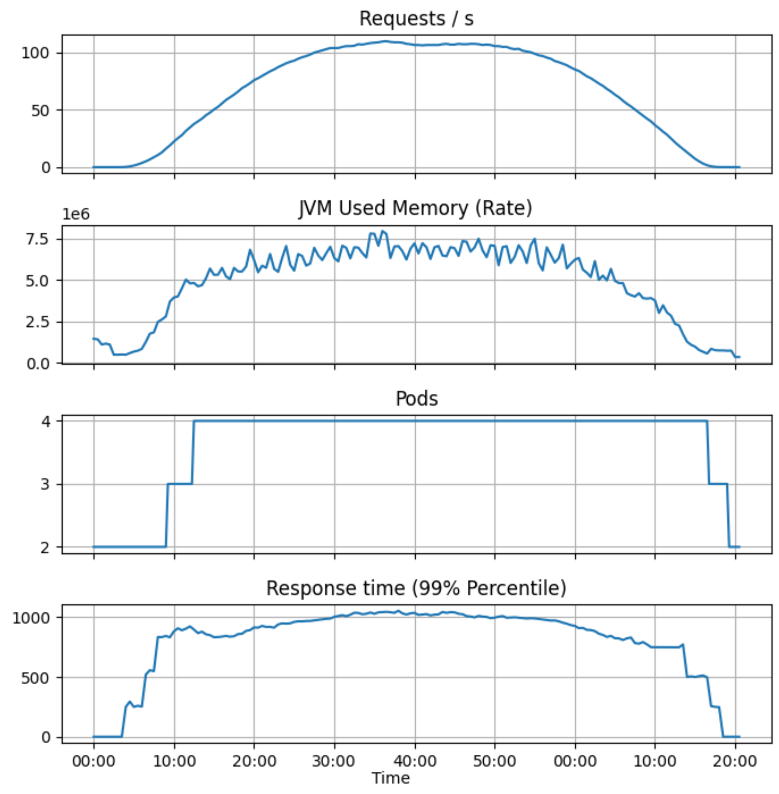


Figure 8.3: Results for Increase-Decrease scenario (Memory-based scaling)

The memory-based scaling approach has the scaler configured to react once the average memory usage of the JVMs on the pods increases by more than 3MB in three minutes, resulting in the scaling behavior seen in figure 8.3. Immediately striking are the very similar shapes of the request and memory usage curves, which is confirmed by a Pearson correlation coefficient of $r = 0.96$. While initial tests using the total average JVM memory usage returned poor results, switching to Prometheus' *rate()* function for the memory metric drastically improved the accuracy of memory as a performance indicator.
Reactions from the scaler come quickly, with an Eagerness of 25 seconds and a pod start-up time of 1 minute and 45 seconds. After the test, the deployment returned to a minimal configuration again with a Plasticity of 2 minutes and 20 seconds. While the initial reaction of the scaler is impressive, the accuracy of the memory metric mostly shines in the early scale-down reaction, which is made possible by the slow decrease of the memory metric towards the end of the test. The behavior of the response time is a mix between the two previous scaling approaches, with a strong initial mean time to repair of 1 minute and 45 seconds, but additional failure periods towards the peak of the tests, resulting in a total failure time of 16 minutes and 40 seconds.

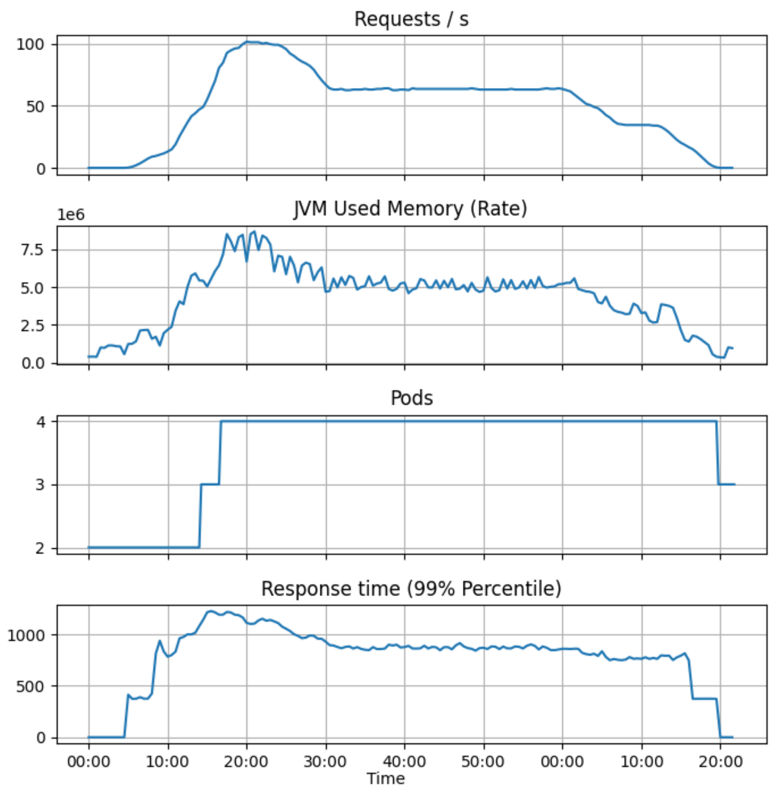


Figure 8.4: Results for Spike scenario (Memory-based scaling)

Accuracy of the memory usage metric is visible again in figure 8.4, with the spike clearly showing in the RAM usage increase of the JVM, resulting in a Pearson correlation coefficient of $r = 0.96$. This is very similar to the correlation in the Increase-Decrease scenario and could come from the fact that the tested application is known to be very RAM dependent.
Behavior of the scaler is also similar to the Increase-Decrease scenario, with an Eagerness of 30 seconds combined with a pod start-up time of 1 minute and 30 seconds. With 3 minutes and 30 seconds, Plasticity is higher than before, seemingly profiting less from the early decrease of the target memory metric towards the end of the load distribution. The user experience is strongly impacted by the spike, with a mean time of repair of 11 minutes and 30 seconds, resulting from the eager but conservative scale-up operation.

8.1.3 Response time-based scaling

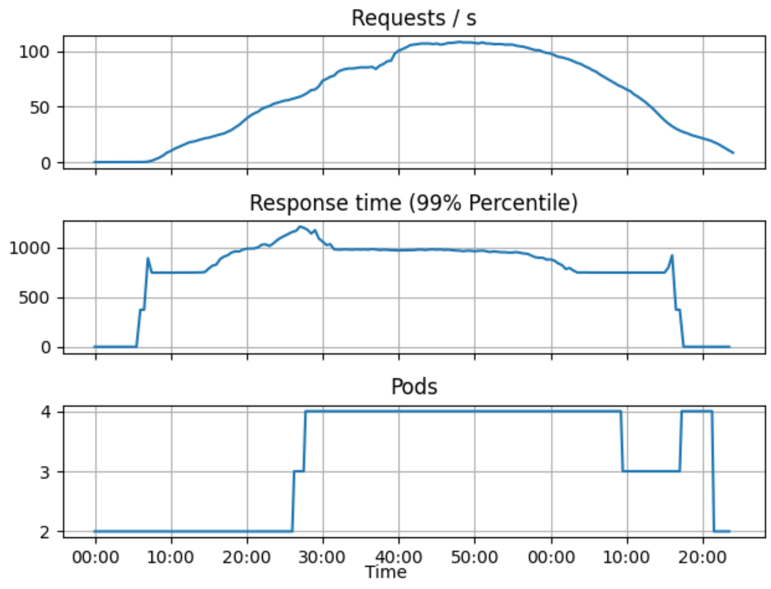


Figure 8.5: Results for Increase-Decrease scenario (Response time-based scaling)

The test results shown in figure 8.5 show the HPA's behavior during the IncreaseDecrease scenario when using a response time threshold of 1s for 99% of requests. The Pearson correlation coefficient between the 'Requests / s' and 'Response time (99% Percentile)' data sets equals $r = 0.65$. While the metric clearly correlates with the system load, the sharp increases and decreases during the transition from and to no load do not reflect the actually much slower increase of system load.
Due to the very conservative scaling approach, with the scaling threshold matching the service-level objective, the additional pods arrive late, with an Eagerness of 3 minutes and average pod start-up time of 1 minute and 55 seconds. After load subsides, the deployment returns back to a minimal configuration with a Plasticity of 5 minutes. For the user, this scaling behavior results in a mean time to repair of 9 minutes and 10 seconds, during which the response time is larger than 1 second for 99% of requests. The oscillation effect visible in the pod count only occurred in one out of the three test runs, so it was likely caused by a pod failure and not by the scaler.

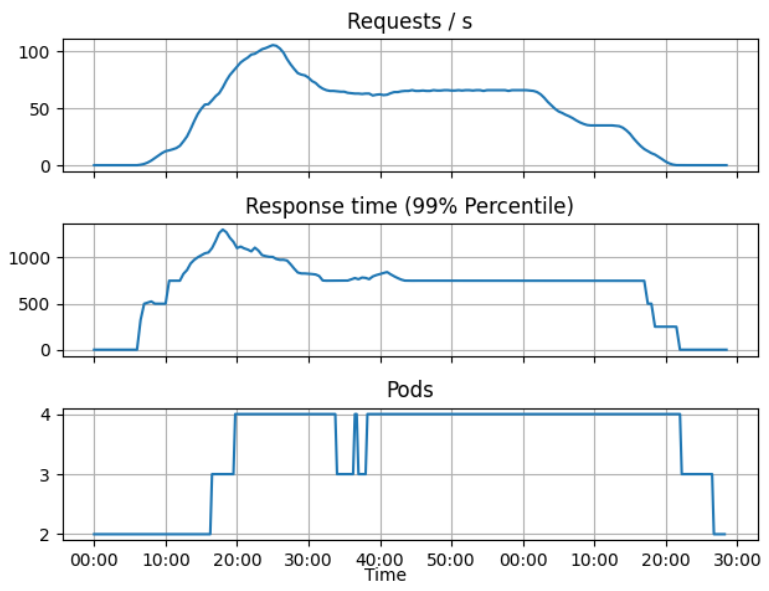


Figure 8.6: Results for Spike scenario (Response time-based scaling)

For the Spike scenario, the accuracy of response times as a performance indicator was much higher, with a Pearson correlation coefficient of $r = 0.82$, which is also clearly visible when graphing the metrics together. Since the load distribution in the Spike scenario rises and falls much more rapidly than in the Increase-Decrease scenario, the behavior of the resulting response times fits much better. Looking over at scaling performance, the sharper increase in response times also leads to a more eager reaction of the scaler, with an Eagerness of 45 seconds, which is slightly dampened by the pod start-up time of 1 minute and 45 seconds. After the requests stopped, the scaler reduced the number of pods to the minimum again with a Plasticity of 5 minutes and 5 seconds. The mean time to repair equals to 7 minutes and 30 seconds, with a notable effect of the additional pods also visible in figure 8.6. Again, oscillation effects are visible in the pod counts, which this time occurred during all three tests. The cause for this might be the drastic drop in response times after the spike, triggering an overly aggressive downscale operation in the scaler.

8.2 Discussion of scaling metrics

When trying to find the best metric to describe a system's performance, the core decision that needs to be made is if the metric should be more low-level and hardware focused or on a higher level looking at the user experience itself. To reflect this split in these tests, two classical hardware metrics, CPU and memory, and an important user-level metric, namely response time, were evaluated and tested in different scenarios.

8.2.1 CPU-based scaling

The CPU metric was overall the most inaccurate, since it shot up quickly after the test start and then fell in a similar rapid fashion after the test ended. The immediately obvious benefit of this is an early reaction of the scaler, which can also be seen in the evaluation when comparing the time intervals between the test start and the first scale up operation. At the same time, this quick reaction also creates an immediate risk of over provisioning resources, as only little load is needed to trigger a substantial reaction by the scaler.
One core issue when dealing with hardware related measures that is also apparent here is the strong dependency on the input. The high CPU usage in these tests arises from the fact that the test method has a very high CPU usage, thus driving the metric up even when there is very little load. On the contrary, if the test method used very little CPU resources but was more I/O-bound, the results could have been flipped, with little to no variation in CPU usage and no change in the metric, even if the systems it depends on, like databases, were under heavy load.
In a real world scenario with a diverse range of different requests, using CPU usage as a reliable performance indicator requires a deep understanding of how the system works and how CPU utilization is affected by different amounts and types of work to build a reliable scaler on top of it, even if the one shown in these tests produced good results.

8.2.2 Memory-based scaling

The second hardware-related performance metric on the other hand has shown an impressively high accuracy in representing the current system load, with by far the highest correlation coefficient between the number of incoming requests and the rate of memory usage by the JVM. The rather low threshold of +3MB per second lead to a more conservative scaling behavior compared to the CPU-based scaler but still a more eager and oscillation-resistant scaler when put side-by-side with the response time-based approach. The result of this is a slightly smaller amount of over-provisioning, which could be mitigated further by modifying the threshold, as the high performance accuracy of the underlying metric allows for easier fine-tuning compared to the CPU approach.
Though similarly to that approach, the memory rate shows a high dependence to the input data, which also explains its high accuracy. Since the payload sizes for the tests were not randomized to ensure that the tests stay reproducible, the only other factor determining the memory usage was the amount of requests being produced, which then finally results in two very similar curves that only differ by a constant factor, being the amount of memory needed to process each request. The oscillation effects seen in the memory curve are a result of the JVM garbage collector and did not affect the metric or scaling behavior significantly.
In summary, the memory-based scaling approach fares better than its CPU counterpart, but suffers from the same caveats by being highly dependent on the input data and application implementation, meaning that an effective use as a scaling metric is only possible in combination with a deep understanding of the underlying application and its handling of resources.

8.2.3 Response time-based scaling

Serving as the core indicator of actual user experience, the response time metric differs greatly from the other scaling approaches by focusing on the real end results of the request processing and scaling work. Apart from the spikes at the start and the end of the test periods, the response-time metric has also proven highly accurate when representing the load put on the system. Being the most conservative of the three approaches, with a response time threshold of 1 second, this scaling configuration lead to a late but swift reaction by the scaler, resulting in short periods of failure but overall very conservative resource usage.
One important point to note is that this is the only tested metric that is visibly affected by the results of the scaler, which is an important property in practice. On the one hand, it prevents an endless scale-up chain triggered by continuous violations of the scaling metric thresholds, which was mostly prevented in the other tests by confining the scaler to an interval of two to four pods in which to operate. On the other hand, this can lead to the danger of oscillation, which occurred in both and only the tests using the response-time based scaler. While not detrimental to the overall scaling performance, it could become a bigger issue in long-term production use with more diverse load scenarios. Fortunately, newer versions of Kubernetes [CFG] provide more granular configuration options for the scaling behavior, like custom step sizes or scale-up cooldowns, which could come in use here.
Finally, the main advantage of using an application level metric like the response time is its weaker relation to the concrete input data and the hardware usage itself. While both the behavior of CPU and memory can be strongly influenced by the type of request, making them more difficult to use in a practical scenario with diverse request parameters, the response time provides a more distanced and aggregated view on the system performance. Additionally, for this concrete case, it holds much more relevance in a practical use case, as response time is usually contractually constrained by service-level objectives and therefore needs to be precisely monitored and controlled. This fact alone makes a strong case for using it as a scaling criteria for the HPA, especially since the response time is strongly affected by scaling operations and thus provides a direct feedback to the scaler.

8.3 Discussion of scaling performance

Apart from the observations reported in the previous section, there are also conclusions that can be made about the scaling implementation overall. Firstly, the architecture as described in section 6.2 performed very well overall. The Prometheus adapter was easily integrated in the existing cluster setup and provided a light-weight connection between the Prometheus metrics system and the Kubernetes metrics API. Extensive configuration possibilities for the adapter allowed access to all existing performance metrics and the easy creation of new ones trough PromQL. Together with the similarly simple deployment of the corresponding HPA, an automatic deployment and configuration of the HPA together with the adapter seems feasible.
On the other side, looking at the end results of the scaling tests makes apparent that the differences between using different metrics and scaling rules are much smaller than anticipated. While clear differences in the accuracy of the metrics themselves can be seen, the behavior of the scaler and consequently the end user experience did not change significantly between the tests and was mostly affected by the scenarios themselves. The core practical difference of the different approaches lies in their reliability in practical scenarios, for which the response time-based approach served as the

most stable performance indicator due to its distance to implementation and hardware details and its visible reaction to actions by the scaler.

8.4 Limitations

As this evaluation took place in the context of a case study in a practical scenario, the amount of limitations and considerations is significantly greater compared to a controlled lab study, which is why the context holds significant weight when interpreting these results.
Firstly, while efforts were made to provide a stable and consistent technical environment for these tests, for example by running each test multiple times, they no less took place in a production cluster also hosting other applications and services in parallel to the ones examined in this thesis. Therefore, performance might have been affected by outside influences not mentioned here, for example by applications running on the same nodes as the pods used in these tests. Additionally, the examined applications are not fully idle when no requests are coming in, as different background routines and batch tasks are run, which are required by the application to operate correctly.
Secondly, the scale and possibilities provided by the testing environment are limited in nature, resulting in a high amount of external control for these tests. Most significantly, this includes the low amount of available resources consisting of four pods and the highly mocked testing method, which doesn't take advantage of many of the different features available in the tested application. As a result of this, generalizations of the observed results should be taken with care and with an understanding of the surrounding context. While deciding for a specific concrete scaling approach on the basis of the given results is difficult, the general implementation and proof of concept for a scaling implementation in a production environment can be used as a guideline for similar solutions.

9 Conclusion

9.1 Summary

The goal of this thesis is to explore the different possibilities for auto-scaling solutions in a practical real world scenario at AEB SE. Motivation for this is their ongoing migration of their application servers, which for the longest time ran on regular virtual machines and were managed almost entirely by hand. To modernize their approach and to mitigate a number of ongoing issues, including difficult cluster management and slow scaling, they are moving their applications to a Kubernetes cluster. The practical goal of this thesis is to collect a list of requirements for this new scaling solution, to use these insights to find accurate metrics for monitoring system performance and to detect scaling needs and to finally implement a proof-of-concept scaling solution using the gathered knowledge and state-of-the-art technologies and methods.
Serving as the technical foundations for the conducted research, a number of different technologies and tools were used to execute the developed ideas and to evaluate them. First on that list is Kubernetes, serving as the basis for the entire technical framework, providing a way to host applications in a containerized and standardized way and to manage them efficiently using a powerful API and different clustering concepts, like deployments, namespaces and pods. A Kubernetes cluster already running at AEB hosts the deployment examined in this thesis. The deployment runs in its own namespace and consists of an operator pod for administrative tasks and two application pods running the actual engines themselves.
To collect performance data from the cluster, the monitoring framework Prometheus is already configured and in use. Prometheus extracts the necessary metrics from different cluster components and then provides outside access to them via a query system using PromQL. One of these tools that takes use of this is Grafana, which turns the raw numerical data into more intuitive visual representations.
For evaluation and tests, Apache JMeter is used to perform load tests on specific applications. It provides an expansive set of tools to interact with the examined system, for example via HTTP requests, and also to evaluate the responses from the system, containing data like error codes or response times.
Requirement elicitation was performed through the form of semi-structured interviews with a wide range of people working at AEB SE. This includes developers and architect with a focus on the applications and their structure themselves, but also IT admins and support staff with different views and expectations for an auto-scaler. When asked about the most important qualities of a good auto-scaling solution for their application, most interviewees put a strong focus on quick response times and high availability for the automated parts of their system, mainly web requests coming from connected customer systems. In the area of configuration options, both a high degree of manual intervention into the scaling behavior and the option to have a self-adapting scaler were mentioned. Moving onto performance indicators, CPU and memory usage stood out at the main hardware related metrics, with job queue length and request response times being the most important application

related metrics. While performance is monitored by the IT staff, it is otherwise not looked at too often, with the option to manually add additional application instances if significant performance decreases are noticed or, in predictive cases, expected. When asked about the current scaling solution, most interviewees expressed their satisfaction, although a more automated approach would be favored by many.
Serving as the object of examination, a test application containing the basic framework functions used in production software at AEB was expanded to include a REST endpoint, serving a function for putting a specific amount of load on the system by sending a request to it with the corresponding parameters. To execute the planned scaling solution, an architecture was developed, which contains different components necessary for the process. Firstly, Prometheus scrapes the pods for performance metrics, which are then queried by the Prometheus Adapter, a special pod running in the same deployment that aggregates and exposes the collected data to the Kubernetes Metrics API. From there, the Kubernetes Horizontal Pod Autoscaler fetches the relevant metrics, compares them with the thresholds and rules defined in its configuration, and if necessary sends a corresponding scaling prompt to the deployment, which then finally adds or removes pods to scale the application deployment either in or out.
The three chosen metrics for the evaluation are the average CPU utilization, the increase rate of the used memory of the JVM and the 99% percentile of request response times. The Prometheus Adapter collects and exposes the metrics, while the HPA has three separate configurations, each listening to one of the listed metrics, with thresholds of 160% CPU utilization, a 3MB memory increase and a maximum response time of 1 second respectively. These three scaling approaches are then put under test using two different load scenarios, a relaxed wave-like scenario and a more extreme spike scenario, executed in practice by JMeter with a maximum of 80 users over a span of 60 minutes each.
Examining the resulting metrics and performance data shows that both the response time and the memory usage give accurate representations of current system load, with CPU utilization losing accuracy due to the high CPU usage of the testing methods. The accuracy of the memory metric can be explained in a similar fashion, concluding that both hardware metrics are highly dependent on the input data and application implementation, making them harder to use reliably in practice, with the response time metric providing a more aggregated view. In terms of scaling performance, both memory usage and CPU utilization tend to overprovision resources, with the latter reacting very quickly even to small load changes. The approach using the response time as a scaling threshold returned more conservative results, with the scaling operations also visibly affecting the scaling metric itself, but suffering from small oscillation effects as a consequence of that. All together, the scaling performance between different approaches did not differ drastically though, the long pod start up time being a shared limiting factor. Due to the limited resources and high degree of mocking in the test methods, these results should be taken with care though and only provide general indications of which approaches could work in practice.

9.2 Future work

While this work hopefully provides some interesting insight into how a state-of-the-art auto-scaling approach can be realized in a practical scenario, there are many aspects that could be explored further in a similar future case study.
First of all, one major limiting factor for the reliability of the evaluation results are the limited

resources available at the time of the research. Thus, in a context where the migration towards a fully containerized cluster using Kubernetes and an optimized architecture has already been completed, the described scaling solutions could be reevaluated on a larger scale, using more refined testing methods, and testing a more diverse range of scaling techniques and load scenarios.
Secondly, one point that was also mentioned in the interviews which was not looked into here is the method of vertical scaling and how it could be used in conjunction with existing scaling methods. For example, the tested applications were largely memory-bound in their performance, so a vertical memory scaling technique could be applied to increase overall performance without needing to allocate additional resources and go through the processes attached to that.
Building on top of the previous point, another aspect that has come up during this research is the possible negative effect of additional pods. For example, the start-up phase of the tested application causes a high amount of CPU usage, which could in turn affect overall average CPU utilization in a manner not expected by the scaler. On top of that, existing bottlenecks like databases could lose even more performance when additional instances are accessing it, actively reducing overall performance through new pods.

Bibliography

[AHM] Kubernetes SIGs. *Helm chart for the Prometheus Adapter*. URL: https://github.com/prometheus-community/helm-charts/tree/main/charts/prometheus-adapter (cit. on p. 39).

[BBD+14] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, S. Krstić. “Towards the formalization of properties of cloud-based elastic systems”. In: *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*. 2014, pp. 38–47 (cit. on pp. 26, 43).

[CFG] Kubernetes. *Configurable scale up/down velocity for HPA*. URL: https://github.com/kubernetes/enhancements/blob/master/keps/sig-autoscaling/853-configurable-hpa-scale-velocity/README.md (cit. on pp. 39, 52).

[CP17] E. Casalicchio, V. Perciballi. “Auto-scaling of containers: the impact of relative and absolute metrics”. In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2017, pp. 207–214 (cit. on pp. 23, 24).

[CTG] Blazemeter Inc. *Custom Thread Groups Plugin for JMeter*. URL: https://jmeter-plugins.org/?search=jpgc-casutg (cit. on p. 41).

[DDB09] M. D. De Assunção, A. Di Costanzo, R. Buyya. “Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters”. In: *Proceedings of the 18th ACM international symposium on High performance distributed computing*. 2009, pp. 141–150 (cit. on pp. 24, 25).

[EC2] Amazon. *Amazon Elastic Compute Cloud*. URL: http://aws.amazon.com/ec2/ (cit. on p. 26).

[GFA] Grafana Labs. *Grafana*. URL: https://grafana.com/ (cit. on p. 17).

[GLG] Gatling Corp. *Gatling*. URL: https://gatling.io/ (cit. on p. 21).

[HKR13] N. R. Herbst, S. Kounev, R. Reussner. “Elasticity in cloud computing: What it is, and what it is not”. In: *10th International Conference on Autonomic Computing ({ICAC} 13)*. 2013, pp. 23–27 (cit. on pp. 11, 25).

[HPA] Kubernetes. *Horizontal Pod Autoscaler*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (cit. on p. 15).

[JLNY04] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, A. F. Yassin. “A practical guide to the IBM autonomic computing toolkit”. In: *IBM Redbooks* 4.10 (2004) (cit. on p. 24).

[JMR] Apache Software Foundation. *JMeter*. URL: https://jmeter.apache.org/ (cit. on p. 19).

[K8S] Kubernetes. *Kubernetes*. URL: https://kubernetes.io/ (cit. on pp. 15, 23, 24).

[KFB18] F. Klinaku, M. Frank, S. Becker. “CAUS: An elasticity controller for a containerized microservice”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 93–98 (cit. on pp. 23, 24, 41).

[LNS] Mirantis Inc. *Lens*. URL: `https://k8slens.dev/` (cit. on p. 16).

[MH11] M. Mao, M. Humphrey. “Auto-scaling to minimize cost and meet application deadlines in cloud workflows”. In: *SC’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2011, pp. 1–12 (cit. on p. 25).

[MKF10] P. Marshall, K. Keahey, T. Freeman. “Elastic site: Using clouds to elastically extend site resources”. In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE. 2010, pp. 43–52 (cit. on pp. 24, 25).

[MKS07] H. A. Müller, H. M. Kienle, U. Stege. “Autonomic computing now you see it, now you don’t”. In: *Software Engineering*. Springer, 2007, pp. 32–54 (cit. on p. 24).

[NCCA14] M. A. Netto, C. Cardonha, R. L. Cunha, M. D. Assunçao. “Evaluating auto-scaling strategies for cloud computing environments”. In: *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE. 2014, pp. 187–196 (cit. on p. 25).

[NFX] Netflix. *Scryer*. URL: `https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270` (cit. on pp. 26, 42).

[PMA] Kubernetes SIGs. *Prometheus Adapter for Kubernetes Metrics APIs*. URL: `https://github.com/kubernetes-sigs/prometheus-adapter` (cit. on p. 38).

[PMS] The Linux Foundation. *Prometheus*. URL: `https://prometheus.io/` (cit. on p. 17).

[PQL] The Linux Foundation. *PromQL*. URL: `https://prometheus.io/docs/prometheus/latest/querying/basics/` (cit. on p. 17).

[WAZ] Microsoft. *Windows Azure*. URL: `http://www.windowsazure.com` (cit. on p. 26).

All links were last followed on May 19, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 31.5.2021, [illegible]

place, date, signature