Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis

Visualizing and Explaining the Scaling Behavior of Self-Adaptive Microservice Systems in Kubernetes

Tobias Rodestock

Course of Study: Softwaretechnik

Examiner: Prof. Dr.-Ing. Steffen Becker

Supervisor: Sandro Speth, M.Sc.

Commenced: July 26, 2021

Completed: January 26, 2022

Abstract

Context. Modern microservice systems are made self-adapting to help manage increasing complexity.
Problem. Self-adapting microservice systems offer little insight in their scaling behavior. This makes debugging and verification of their behavior difficult and time consuming.
Objective. Explore how to offer better insight and visualization of the scaling behavior of self-adaptive microservice systems. Detect anomalies in the behavior and report them to an issue management system.
Method. Setting up a test system by making an existing microservice system self-adaptive and developing a concept and implementing a prototype that can visualize and explain the scaling behavior of the test system.
Result. A useful concept for visualizing the autoscaling behavior of a self-adaptive microservice system has been created. In the evaluation many experts state that it helps to solve different problems they face when trying to understand the autoscaling behavior of a self-adaptive microservice system.
Conclusion. The developed concept is a good starting point for further improvements in visualizing and explaining the autoscaling behavior of a self-adaptive microservice system.

Kurzfassung

Kontext. Moderne Microservice-Systeme werden selbstanpassend gemacht, um die zunehmende Komplexität zu bewältigen.
Problem. Selbstanpassende Microservice-Systeme bieten wenig Einblick in ihr Skalierungsverhalten. Das macht das Debugging und die Verifizierung ihres Verhaltens schwierig und zeitaufwendig.
Ziel. Erforschen, wie man einen besseren Einblick und eine bessere Visualisierung des Skalierungsverhaltens von selbstanpassenden Microservicesystemen bieten kann.
Methode. Aufsetzen eines Testsystems, indem ein bestehendes Microservice-System selbstanpassend gemacht wird. Die Entwicklung eines Konzepts und die Implementierung eines Prototyps, der das Skalierungsverhalten des Testsystems visualisieren und erklären kann.
Ergebnis. Ein nützliches Konzept zur Visualisierung des Autoskalierungsverhaltens eines selbstanpassenden Microservice-Systems wurde erstellt.
Fazit. Das entwickelte Konzept ist ein guter Ausgangspunkt für weitere Verbesserungen bei der Visualisierung und Erläuterung des Autoskalierungsverhaltens eines selbstanpassenden Microservice-Systems.

Contents

1 Introduction **1**

2 Foundations and Related Work **3**
2.1 Foundations . . . 3
2.2 Related Work . . . 7

3 Concept **11**
3.1 Overview of the Concept . . . 11
3.2 Exporting and Processing Scaling Events . . . 11
3.3 Visualizing System State . . . 15
3.4 Analyzing for Performance Problems . . . 17
3.5 Reporting Issues . . . 18

4 Implementation **23**
4.1 Implementing the Test System . . . 24
4.2 Implementing the Prototype . . . 30

5 Evaluation **35**
5.1 Goal Question Metric Approach . . . 35
5.2 Expert Survey . . . 36
5.3 Discussion . . . 39
5.4 Threats to Validity . . . 40

6 Conclusion **43**
6.1 Summary . . . 43
6.2 Limitations . . . 43
6.3 Future Work . . . 44

Bibliography **45**

A Artifacts of the Expert Review **47**
A.1 English Presentation and Questionnaire . . . 47
A.2 German Presentation and Questionnaire . . . 47
A.3 Load Profile . . . 47
A.4 Questionnaire Results . . . 47

List of Figures

3.1 Overview of the concept. 12
3.2 Flowchart of the Concept. 12
3.3 Example transformation. 14

4.1 Implemented system. 23
4.2 Visualization of the t2-store-random-infinite.jmx load profile. 27
4.3 The new load curve. 28
4.4 List of events. 32
4.5 List of sets of events. 32
4.6 Screenshot of the dashboard. 33
4.7 Metrics query page. 34
4.8 Gropius ID page. 34

5.1 Average rating for **Problem 1**. 38
5.2 Average rating for **Problem 2**. 38
5.3 Average rating for **Problem 3**. 38
5.4 Average rating for **Problem 4**. 39
5.5 Average rating for **Problem 5**. 39

Acronyms

APPD Automatic Performance Problem Detection.

CLI Command Line Interface.

HTTP Hyper Text Transfer Protocol.

ID Identity.

JSON JavaScript Object Notation.

PromQL Prometheus Query Language.

UI User Interface.

1 Introduction

Microservice architectures are getting increasingly complex. With increasing complexity management of these system also gets more difficult. To alleviate this problem systems are made self-adapting. This means systems can adapt themselves by reconfiguring their architecture to react to changes in its environment. For example they can scale individual services to cope with changing system load. But this leads to a lack of insight into the adaption decisions, which is needed for debugging, optimization, and validation of the system. This lack of insight was also noted by many experts in the review of the concept of this thesis. The objective of this thesis is to explore ways to visualize and explain the autoscaling behavior of self-adaptive microservice systems. This leads to the following research question:

> **RQ 1**
> *How can the autoscaling behavior of self-adaptive microservice systems be visualized and explained?*

First, a concept for visualizing and explaining the autoscaling behavior of self-adaptive microservice system is created. Then a test system is set up using an existing microservice system. The test system is used to generate scaling events. The concept and the developed prototype are evaluated to gain valuable feedback. First the experts are introduced to the problem area and are presented the concept and the prototype. Then a survey is completed to evaluate if the concept and the prototype help to solve issues developers have understanding the autoscaling behavior of a self-adaptive microservice system.

The experts appreciate the developed concept and the prototype. Many experts state that the concept solves issues they encounter when working with self-adapting microservice systems. Valuable improvement ideas are given by the experts of the evaluation. All experts agree that they would use a fully implemented version of this concept.

The main contribution of this thesis is a concept for visualizing the autoscaling behavior of a self-adaptive microservice system, a implemented prototype and the evaluation of the concept and the prototype through an expert survey.

Thesis Structure

This thesis is structured as follows:

Chapter 2 – Foundations and Related Work: In this chapter the foundations for this thesis is explained. Related work and the research methodology are shown.

Chapter 3 – Concept: This chapter explains the developed concept.

Chapter 4 – Implementation: In this chapter first the set up of the test system is described, then the design and implementation of a prototype based on the concept.

Chapter 5 – Evaluation: The concept and the prototype are evaluated with a expert survey. The design, realization and the results of the evaluation are discussed in this chapter.

Chapter 6 – Conclusion In the final chapter the thesis is summarized and possible future work is highlighted.

2 Foundations and Related Work

This chapter outlines concepts and tools relevant to this thesis. In Section 2.1 the foundation for understanding this thesis are describe. The process of researching foundations and related work is shown in Section 2.2.1 .Related work is introduce and discussed in section 2.2. .

2.1 Foundations

Self adaptive systems

A self adaptive system is a system that can change its behavior or structure according to changes in the environment, the system itself or the goals of the system [LR04]. Self-adaption covers a wide range of properties [IBM05]:

- Self-configuring: A self-configuring system can adapt to changes in its environment according to predefined policies. This can include adding or removing components or even changing the structure of the system.
- Self-healing: The system can detect and fix malfunctions on its own. This helps the system to get more resilient.
- Self-optimizing: A self-optimizing system can change the allocation of resources dynamically according to changes in the workload the system is experiencing.
- Self-protecting: The system can detect and respond to security threats from anywhere. It can take actions to mitigate its own vulnerabilities.

Autoscaling

Autoscaling, as part of the self-optimizing properties of self adaptive systems, automatically adapts the amount of allocated resources in cloud computing [New21]. There are two different types of scaling: The first is vertical scaling, which means that the resources available are increased. For example. a server receiving more CPU power or more memory. The second type of scaling is horizontal scaling. The amount of resources is increased or decreased by, for example, using three machines instead of one. Now autoscaling means that the scaling is done automatically based on changes in the environment, for example an increase in system load.

Explanation

Bohlender et al. [BK19] define an explanation as follows:

Characterization 1 (Explanation)
A representation E of some information I is an explanation of explanandum X with respect to target group G iff the processing of E by any representative agent A of G makes A understand X.

The scope of an explanation depends on the recipient of an explanation described as agent A of a target group G in the above definition. This means that depending on the knowledge of the target group, the depth of an explanation needs to vary.

An explanation has a cost attached to it. The cost of an explanation is high when the time the processing of the explanation through an agent is close or higher than the time saved by providing the explanation. Contrary, the cost of an explanation is low when the time to process the explanation is significantly lower than the time saved.

Microservice architecture

Microservice architecture is a way to develop single applications within a group of services, whereas each service runs an independent process. The services communicate with each other through a lightweight framework, for example using a HTTP resource API. Each service is independently deployable by an automated commanding unit. A system developed in the described microservice architectural style is called microservice system [Fow] [New21].

Kubernetes

Kubernetes[1] is an open source container orchestration tool. It is one of the leading solutions for managing, deploying and scaling containerized applications. The user can interact with the Kubernetes Cluster through the Kubernetes API. Kubernetes Resources are described in YAML or JSON format and are afterwards sent to the Kubernetes API to create or configure them.

The smallest unit in Kubernetes is a pod. A pod is a group of one or more containers. A pod consisting of a single container is the most common use case. A replicaSet can be used to guarantee that a certain number of pods is available at all times.

Deployments are used to declaratively manage pods and replicaSets. In a deployment the desired state of pods and replicaSets is configured. Kubernetes ensures that the state of the required resources always matches the desired state of resources through controlled changes. Deployments can be automatically scaled using the Horizontal Pod Autoscaler.

Services in Kubernetes are used to describe how a group of pods or a deployment can be accessed. A service can be configured to only be accessible from within the cluster, from the outside of the cluster, or it can be used for load balancing purposes.

[1] https://kubernetes.io/

Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler[2] is a tool built into Kubernetes that can scale selected workload resources. As the name suggests, the Horizontal Pod Autoscaler does horizontal scaling and can scale deployments or stateful sets by increasing or decreasing the amount of pods. An increase in pods is triggered when a metric that is used for scaling exceeds its desired value. If all metrics are below their desired values for a certain time period the amount of pods will be automatically scaled down. The amount of pods needed is calculated by dividing the current value of a metric by the desired value and multiplying the result with the current amount of pods. Which results in the following formula:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

The Horizontal Pod Autoscaler can scale on multiple different metrics at the same time. It will calculate the amounts of pods needed for every metric separately and then choose the highest amount of pods needed.

To retrieve the metrics for scaling the Horizontal Pod Autoscaler can access three different Kuberentes API, the `metrics.k8s.io` API, the `custom.metrics.k8s.io` API and the `external.metrics.k8s.io` API. The `metrics.k8s.io` API is implemented by the Kubernetes Metrics Server. The custom and external metrics APIs have to be implemented by a custom solution[3].

Custom Metric Server

The Custom Metric Server[2] is an implementation of the `custom.metrics.k8s.io` and `external.metrics.k8s.io` API. There are several different implementations available. To implement a custom implementation of the Custom Metric Server a boilerplate is provided in the Kubernetes repository.

Helm

Helm is a package manager for Kubernetes applications. The applications can be defined in a Helm Chart. The configuration files for a Helm Chart are written in YAML. Using the Helm chart a predefined application can easily be installed or upgraded on a Kubernetes cluster[4].

Prometheus

Prometheus is an open source monitoring system with a dimensional data model, flexible query language, efficient time-series database, and modern alerting approach [5].

Prometheus can be set up using the `kube-prometheus-stack` Helm Chart.

[2] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[3] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[4] https://helm.sh/
[5] https://prometheus.io/

Extended Berkley Packet Filter (eBPF)

Extended Berkley Packet Filter (eBPF) is an extension for the Linux kernel. It allows running sandboxed programs in the kernel without the need to modify the source code of the kernel. It allows for a wide range of of applications using hooks for system events, network events, function calls and more. Once a hook is triggered the attached eBPF program is run. Some of the use cases for eBPF include performance monitoring, security and tracing[6].

Pixie

Powered by eBPF Pixie is a observability tool for Kubernetes. Without the need to instrument the application it is monitoring it can offer a wide variety of information. It offers resource metrics for pods in the Kubernetes cluster like CPU and memory utilization as well as I/O metrics. It supports tracing for a wide range of protocols like HTTP, Kafka, MySQL and more. Pixie can show the network traffic and flow of a cluster, show JVM metrics and even the application CPU profile via a flamegraph[7].

T2Store

The T2Store[8] is a microservice reference application implementing the Saga pattern [Ric18]. It is implementing a sample web shop selling tea and is loosely based on the TeaStore application. The business logic of the T2Store is implemented with seven services and a additional external services mimicking a payment provider. The T2Store is instrumented with Micrometer and has a configuration for monitoring with Prometheus. It comes with Kubernetes deployment files for the services as well as load profiles for load testing the application.

Cross-Component Issues and Gropius

Speth identified in [Spe19] that current issue management systems are not sufficient to handle the complexity of working with microservice systems. A simple feature request for the microservice system might require multiple changes in the different microservices. Current tools for issue management only allows an issue to be part of one project, where usually one project is used per microservice.

Speth et al. expanded on Speths earlier work and presented Gropius [SBB20] a cross-component issue management system. They also defined a metamodel for cross-component issues in [SBB21]. Gropius allows the user to create a software project, then components can be added to the software project. Each component is linked to the code repository and the issue management system where the component is managed. An issue can be created and all components that are affected by that issue can be added to the issue. Gropius then synchronizes the issue with the issue management systems used for the individual components. This allows users to keep using their existing issue management

[6] https://ebpf.io/
[7] https://px.dev/
[8] https://github.com/t2-project

systems. Speth et al. also created a Gropius extension for Visual Studio Code[9] [SKBB21]. The extension allows the developer to directly manage the cross-component issues without the need to switch to another tool. If explanations for problems with the autoscaling behavior of a self-adaptive microservice system are sent to Gropius, the developer can also see the explanation while he is working on the code.

2.2 Related Work

2.2.1 Literature Research Methodology

Different methods where used to find related work:

First of all, starting from a research paper provided by the supervisor of this thesis snowballing was used to acquire additional input. Two research papers where found from the initial research paper. Going out from these finding to additional research papers where found that are of interest for this thesis. Result where chosen for further examining based on the title of the reference, then the abstract was checked if the work contains relevant information.

The search engine Google Scholar[10] was used to find additional research. The following search terms or combinations of them where used: Autoscaler, autoscaling, cyber physical systems, explanation, explainability, horizontal pod autoscaler, kubernetes, microservice, self-adapting, self-adaptive.

Of each query, the first two sites of search results where checked for relevant work. Results where rejected based on their title and abstract. The remaining research papers where examined more in depth, to see if they contain research relevant to this thesis.

Additionally the Google search engine[11] was used to look for technologies related to the concept of this thesis.

MAPE-K loop

A commonly used feedback loop in the area of self adaptive software systems is the MAPE-K loop [KC03]. It consists of four steps:

- The first step is *Monitoring*. Here data from the software and its environment is collected.
- The second step is *Analyzing*. Here the data gathered through monitoring is analyzed for the need of adaption.
- The third step is *Planning*. Here it is determined how the system needs to adapt to the changes in the software and its environment detected and analyzed in the first two steps.
- The final step is *Executing*. Here the changes that where planned in the third step are applied to the system.

[9]

[10] https://scholar.google.com

[11] https://www.google.com

The MAPE-K loop focuses on how to implement a system to be self adaptive. While the focus of the is thesis will be on exploring how to generate explanations for the behavior of a self adaptive system.

MAB-EX loop

Blumreiter et al. [BGG+19] proposed the MAB-EX loop. Based on the MAPE loop the MAB-EX loop is a framework for self-explaining systems. The MAB-EX loop similar to the MAPE loops consists of four steps:

- The first step is *Monitor*. The environment and also the recipient of the evaluation are monitored.
- The second step is *Analyze*. The data from the monitoring is analyzed, if a explanation is needed. This can be done by identifying irregularities in system behavior or if the user demands an explanation.
- The third step is *Build*. If the need for an explanation is identified, a explanation is generated. To generate the explanation a internal model of the system, called explanation model, is used. The explanation model captures the behavior of the system and the causal relationship between the components of the system. A model can be implemented in different ways like a decision tree or a state machine.
- The last step is *Explain*. Now the explanation generated in the third step is fitted to the recipient. A recipient model is used where the preferences of the recipient and the needed depth of the explanation are described.

The MAB-EX loop focuses more on catering for different recipients of the explanations. This thesis focuses on more experienced developers and technical experts.

Severity

Tsagkaropoulos et al. propose Severity [TVP+21] an approach of scaling microservice systems in an cloud environment. Severity detects situations where quality of service rules are violated. When a violation is detected Severity calculates the severity of the situation and proposes an adaption. While Severity focuses on improving the performance of adaption decisions, the focus of this thesis is to offer better insight into adaption decisions taken by the microservice system.

Comprehensible and Dependable Self-Learning Self-Adaptive Systems

Klös et al. [KGG18] propose a system that can adapt based on timed adaption logic. The adaption rules are improved using a learning algorithm. The system records various information like the system condition, what triggered the adaption and the expected and actual effects of the adaption. All the information gathered is then fed into the learning algorithm to improve the adaption decisions. In [Klö19] Klös mentions that the gathered information can already be used to explain the system behavior or be used to create textual explanations for non-experts.

2.2.2 Current Visualization Options

Kubernetes Dashboard and CLI

The default method for getting insight into the cluster and the auto scaling behavior of a microservice system deployed in Kubernetes is through the Kubernetes Dashboard or CLI. Both can show the current replica size of the deployed services in a list view. The maximum amount of replicas can only be viewed in the configuration of the HPA. In the dashboard it is not possible to get a list of all scaling events that are happening. It is only possible to get the scaling events of a single deployment in the detail page of that deployment. In the CLI it is possible to get a list of all scaling events by filtering for all events by the Horizontal Pod Autoscaler. The only metrics that the Dashboard and the CLI can provide are the CPU and Memory utilization, if the Metric Server is enabled. In summary, the Dashboard and CLI do not provide a good overview over the state of the system in regards to autoscaling. Relevant information is scattered across many places and often poorly displayed.

Datadog

Datadog[12] is one of the leading solutions for monitoring and securing cloud applications. It enables ingesting information from a big range of sources through Datadog integrations. These integrations are available for a wide range of cloud applications. The data can be displayed through configurable dashboards. The big problem with Datadog is that the developers still have to select the metrics they need and build the dashboards themselves. This thesis already selects relevant data and offers a prebuilt dashboard that visualizes the scaling behavior of a microservice system.

[12] https://www.datadoghq.com/

3 Concept

In this chapter we give an overview of the concepts developed in this thesis. In section 3.2 we show how we plan to give a better visualization of system state and behavior. We also show how we reduce the amount of information coming in to a more manageable level. In section 3.3 we describe how we utilize an existing tool to analyze a system for performance issues. In section 3.4 we explain how we analyze the microservice system for performance problems. Finally in section 3.5 we first describe what we need to report an issue to the right place. Then we explain which problems of the system are reported and how the content of the issue is composed.

3.1 Overview of the Concept

The system proposed in this thesis is roughly depicted in Figure 3.1. The proposed system consists of an explanation system that can visualize the autoscaling behavior of a self-adaptive microservice system. The self-adaptive microservice system is deployed inside a Kubernetes cluster. The microservice system is scaled by the Horizontal Pod Autoscaler. The autoscaler gets the metrics it needs for scaling from the Custom Metric Server. Prometheus monitors the microservice system and the Custom Metric Server gets the metrics from Prometheus. In addition a eBPF monitoring system provides the explanation system with metrics to visualize and explain the autoscaling behavior. The concept roughly follows the path of a scaling event inside the explanation system as depicted in Figure 3.2. Once the explanation system gets a autoscaling event from the Horizontal Pod Autoscaler the event is processed. The processed event is then used alongside the metrics provided by the eBPF monitoring system to visualize the autoscaling behavior of the self-adaptive microservice system. Additionally the explanation system monitors the microservice system for performance problems, as well as the Horizontal Pod Autoscaler for issues with the autoscaling. If a problem is detected a cross-component issue [SBB21] is reported to Gropius [SBB20].

3.2 Exporting and Processing Scaling Events

3.2.1 Exporting the Kubernetes Events

The first step for visualizing the system state and explaining scaling behavior is exporting the autoscaling-related Kubernetes events, to our explanation system for further processing. As a tool of choice we use the Kubernetes Event Exporter. It offers sending events to multiple different sinks including HTTP endpoints via WebHooks and extensive filtering options.

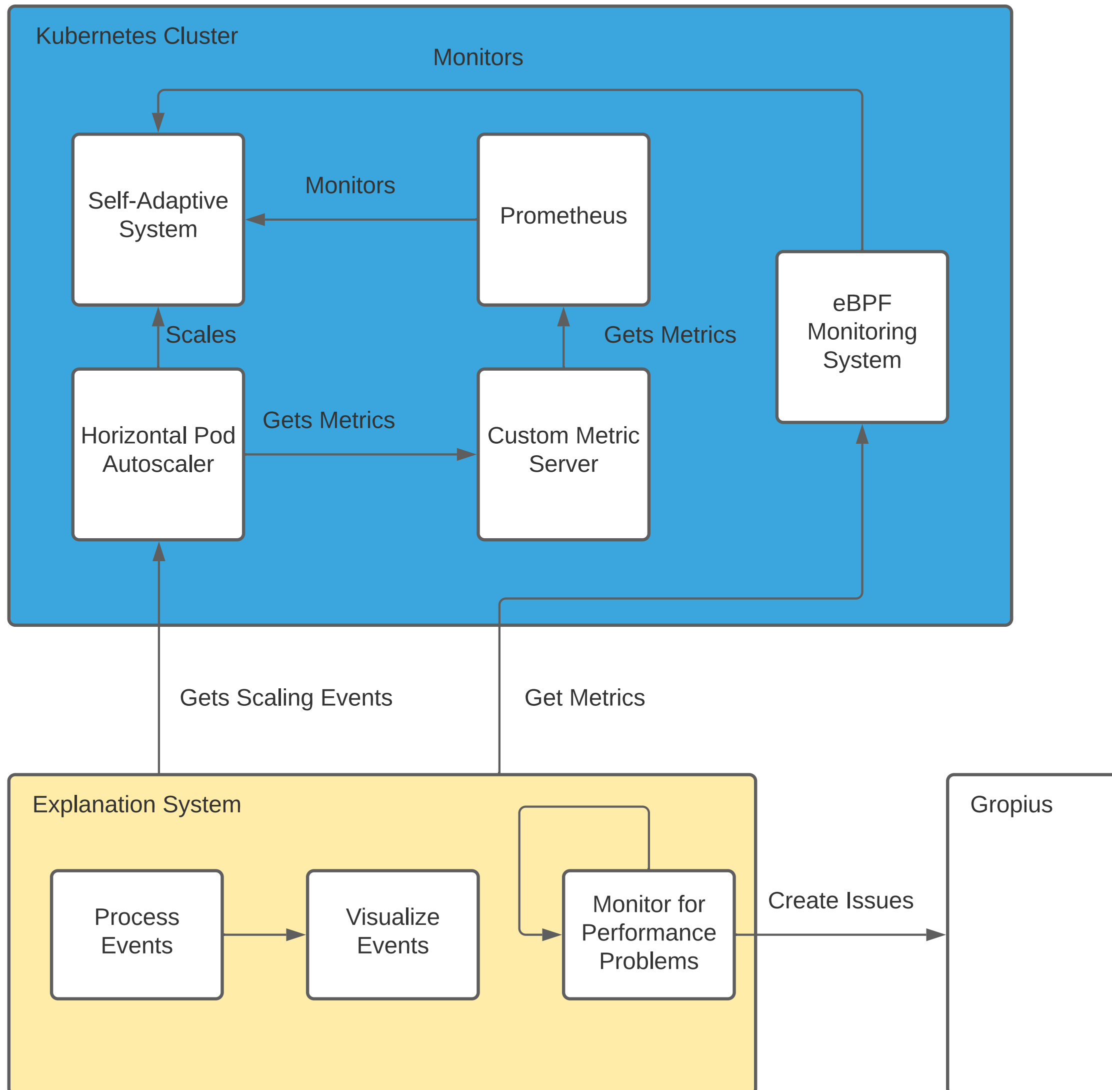

Figure 3.1: Overview of the concept.

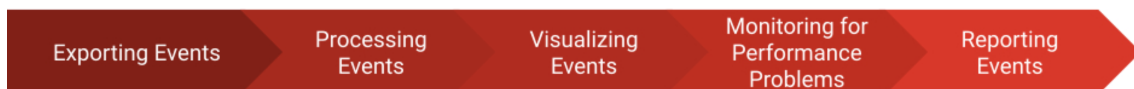

Figure 3.2: Flowchart of the Concept.

There are mainly two different Kubernetes components that emit events related to autoscaling: the Horizontal Pod Autoscaler and the Deployment Controller. Using the `kubectl get events --field-selector involvedObject.name=uibackend` command, yields the events for the uibackend deployment. For a scaling action there are two relevant events.

For the Horizontal Pod Autoscaler:

```
LAST SEEN   TYPE     REASON                     OBJECT
2m5s        Normal   SuccessfulRescale          horizontalpodautoscaler/uibackend
```

```
MESSAGE
New size: 2; reason: external metric traffic above target
```

For the Deployment Controller:

```
LAST SEEN   TYPE      REASON                        OBJECT
2m5s         Normal    ScalingReplicaSet            deployment/uibackend
MESSAGE
Scaled up replica set uibackend-cff658987 to 2
```

Both, the event of the Horizontal Pod Autoscaler and the Deployment Controller, are for the same scaling action that has taken place. But the Horizontal Pod Autoscaler event conveys more information, therefore it is sufficient to only export the events emitted by the Horizontal Pod Autoscaler. There are still more events the Horizontal Pod Autoscaler can emit. These events are emitted when the Horizontal Pod Autoscaler cannot get the metric it needs for scaling or it can not calculate the desired replica size for a deployment. These events are relevant to the creation of issues and will be discussed in detail in section 3.3.1.

Because the event exporter already has a filter function built in, it makes sense to filter the different types of events already in the exporter. The filtered events can then be sent directly to the correct HTTP endpoints. This eliminates the need to implement our own filtering solution.

While the concept of this thesis focuses on Kubernetes and the Horizontal Pod Autoscaler other autoscaling solutions can be supported as well. The scaling decisions of an autoscaling solution must be able to be exported to the explanation system. Then each format needs to be transformed to the same format as the Kubernetes events are. This is for example possible for the Google Compute Engine autoscaler[1], the Microsoft Azure autoscaler[2], and the Amazon EC2 autoscaler[3] as they all emit logs similar to the scaling events of the Horizontal Pod Autoscaler.

3.2.2 Extracting Relevant Information

The exported events need to be further processed after being received by the explanation system. Time and date as well as the scaled component can be directly retrieved from the event. Additional information is included in the message part of the event. There the scaling type, the new replica size and the metric that was used for scaling can be extracted. Taking the example event from earlier, a transformation looks like the following:

[1] https://cloud.google.com/compute/docs/autoscaler
[2] https://azure.microsoft.com/de-de/features/autoscale/
[3] https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html

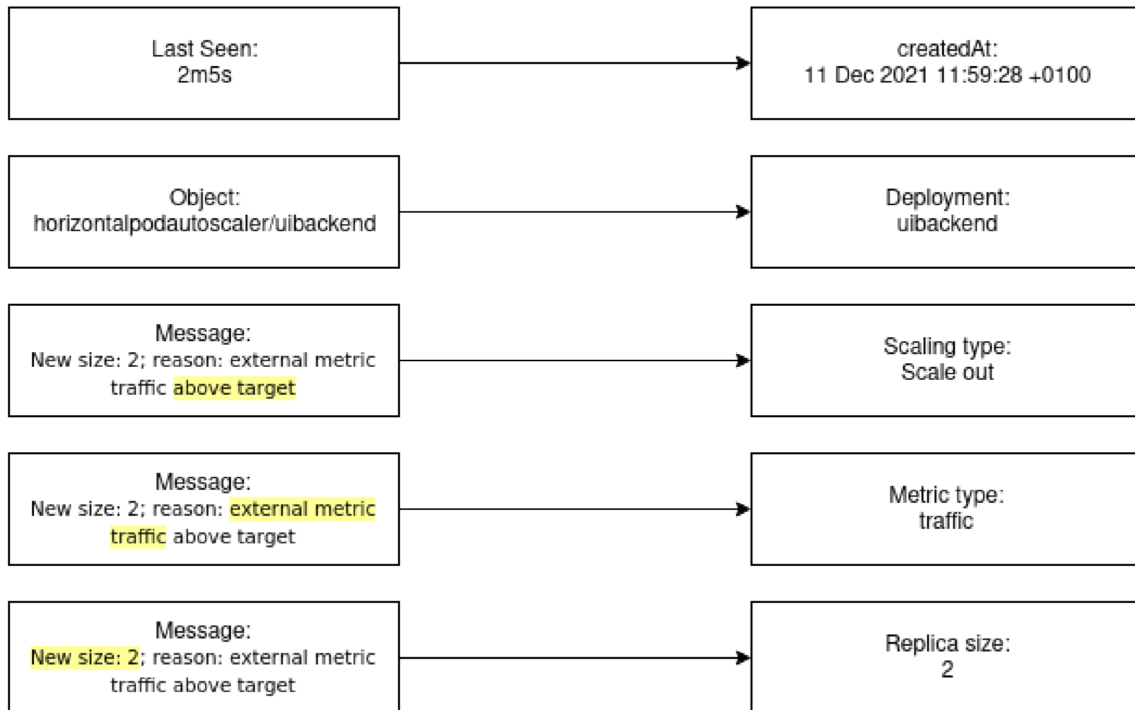

Figure 3.3: Example transformation.

3.2.3 Grouping of scaling decisions

In a micro service system where the system load is volatile, a lot of adaption decision can take place in a short amount of time. This can be a problem considering the cost of explanations. Not the cost of a single explanation for a scaling decision increases, but the cost for explaining the scaling behavior of the whole system increases. To prevent the need for generating a explanation for every scaling decision that takes place, we will need to group related scaling decisions to a set. If all decisions in a set have the same reason, then we only need to generate one explanation for the set of scaling decisions. Thus reducing the amount of explanations and keeping the cost of explaining the scaling behaviour of the system low. For grouping of related scaling decisions we will consider three different types.

Grouping by Scaling Type

The simplest method of grouping scaling decisions is by scaling type. Scaling type here refers to the scaling action taken by the system. For horizontal scaling that refers to be a scale in or a scale out. For vertical scaling that means a scale up or a scale down. We can use the scaling type to determine if scaling decisions are unrelated. That means if the scaling type of two subsequent decision do not match, we can assume that those two do not relate to each other. This eliminates the need to use other methods for grouping.

Grouping by Time

A continuous increase or decrease in system load can lead to the need of multiple scaling decisions. Related events are implied, if multiple scaling processes take place within a small time step. On the other hand, if a longer time period passes between two scaling decisions it can be assumed that they are not related to each other.

Grouping by Derivative

Spikes in system loads followed by sharp drops and subsequently more spikes could lead to scaling decisions being recognized as related by time. To prevent that from happening, we can take a look at the derivative of the metric curve that was used for scaling. If a certain percentage of the derivative between the current and the last scaling decision is negative, then we can conclude that they do not relate to each other.

Combining the Grouping Methods

One of these methods alone is not enough to reliably determine if scaling decisions are related to each other. But if we first check for relation by scaling type, then by time, and finally by derivative and all 3 methods conclude that there is a relation between scaling decisions, we can be reassured that they are indeed related.

3.3 Visualizing System State

3.3.1 Displaying Scaling Events

The sets of grouped events are displayed in a list. Basic information is provided for each set. The information includes date and time of the first and last scaling event in that group, the scaling type, the metric type and the number of events grouped into a set. For each group of scaling events, a detailed view of relevant information will be given. The view contains a textual explanation for the reason for the scaling decisions, basic information about the scaling decisions, graphs for the metrics which the Horizontal Pod Autoscaler uses for scaling, metrics useful for analyzing behavior and the scaling events that are grouped into this set of events. Each of these components is described in more detail in the following paragraphs.

Textual Explanation

Different ways of displaying information are needed depending on the recipient of an explanation. That is why both a textual explanation and a detailed view of the information available for the scaling events is provided. The textual explanation contains the component, the change in replica size with the type of scaling, the change in the metric responsible for the scaling and the change in traffic.

The explanation is generated from a template. An example for such a template is:

```
The <component name> component was scaled <in|out> by <replica change> to a replica size of
<replica size>. The <metric name> metric was used for scaling and <increased|decreased> by <
metric change>% since the last scaling. The traffic <increased|decreased> by <traffic change>%
 in the same time frame.
```

Detailed Information

Showing Changes in Relevant Metrics

The desired replica size is regularly calculated for each metric individually, when a deployment is scaled with the Horizontal Pod Autoscaler on multiple metrics. Then the highest calculated replica size is chosen for scaling the deployment. The information on which metric was used for scaling is readily available as we have already seen. Here we also show what is happening with the other metrics and their desired replica size. To visualize that, a graph is automatically created for each metric showing how the metric changed and how many replicas would have been needed.

To be able to automatically create the graphs for visualizing the behavior of all metrics, we need to know which metrics are used for scaling, a way to retrieve each metric, and the desired values of the metrics to calculate the desired replica size for each metric. Using the Kubernetes API the configuration of the Horizontal Pod Autoscaler can be accessed for each deployment in a cluster. The output consists of the metrics used for scaling as well as their desired values. What we cannot get automatically is the actual metrics values. In this thesis Prometheus is used for monitoring the micro service system and providing the metrics for scaling. That means in the case of this thesis the user needs to add the PromQL queries for each metric, so that the explanation system can retrieve the values for each of them. Having all the information available, the desired replica size for each metric can now be calculated. Here the same formula as the Horizontal Pod Autoscaler is used for calculating the desired replica size.

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Each graph shows the metric on one y-axis and the desired replica count for that metric on the second y-axis.

Traffic and Flame Graph

Pixie offers two types of metrics that can be especially useful for debugging purposes. The first metric is a full view of all HTTP messages coming in and out of the service. Detailed information is provided, for example the status of the HTTP request as well as the full request body can be accessed.

The second metric is a method flame graph where Pixie regularly monitors the stack of the application to determine which methods are called and for how long they are called. The user is provided with the flame graph and a percentage of which methods were used and how long they were used.

Displaying the Grouped Events

In the end, all events that are grouped into a set are listed in the detailed view of that set. Here the information also includes the date and time of the scaling decisions, the scaling type, the metric that was used for scaling and the new replica size.

3.3.2 Concept for the Dashboard

A major problem currently is that the state of the microservice system concerning the scaling is only represented very abstractly through list views in the Kubernetes dashboard or CLI. A visual representation of the system state would aid in understanding the scaling behavior better.

We choose a service dependency graph as the visualization of choice. Each service in the graph has a visualization of the current and the maximum replica size. In addition, the edges between the services can be annotated with the amount of traffic that is going through the edges. The service dependency graph with a baseline of metrics that can represent the load the system is facing are combined. These metrics include CPU, Memory and I/O usage as well as incoming traffic in general and are visualized by a graph for each metric. Finally, the service dependency graph is put in a temporal context with a timeline. The user can scrub through the timeline and the service dependency graph will reflect the status of the system at that time, while the graphs of the metrics show what the system load was at the time selected in the timeline.

The necessary information for the service dependency graph as well as the baseline metrics can be provided using a eBPF monitoring system. This has the added benefit that the service dependency graph and the metrics can be presented for nearly every microservice system regardless of how the application is instrumented or what monitoring solution is set up.

3.4 Analyzing for Performance Problems

Finding the underlying cause for a scaling decision poses a major problem. If a service is scaled but another service has a problem and is the real reason for the scaling the developer will have to debug the application.The Automatic Performance Problem Diagnostics (APPD) [Wer15] approach can be applied to detect and analyze the root cause for software performance problems. It uses an experimental approach where it interacts through adapters with the load generators, instrumentation and monitoring tools. The adapters are used to abstract the APPD approach from the actual tools used for load generation and monitoring. Each experiment has a description of the load that needs to be generated, where the application has to be instrumented and which metrics are needed for analyzing. The APPD approach is focused on detecting performance problems in the software development phase. One of the limitations for APPD not working in an production environment is the overhead of the monitoring needed to get accurate monitoring data. This problem is solved by eBPF observability tools like Pixie.

To make the APPD approach work in a live environment, the following changes have to be made. In the original approach predefined experiments are conducted as follows. The experiment plan describes where the application has to be instrumented and which metrics have to be collected.

Then a series of load tests are performed. In the next step the measurement data is fed to APPD together with the instrumentation and load description. Finally APPD uses the measurement to analyze the application for performance problems.

Now for using the APPD approach in a live environment, we first have to sort the provided experiment plans by those that can be satisfied in regards to the measurement data that needs to be collected. Only the experiment plans where the measurement requirements can be satisfied are executed. Therefore an adapter for the instrumentation is not needed. The adapter for the metrics is still needed to feed the APPD the measurement data.

Now another important step for making the APPD approach work in a production environment is to classify the incoming system load. We map the load that is happening to the load profile description we have in the experiment plans. Once we identify that the current system load corresponds to the load description in an experiment plan we can feed the load description together with the instrumentation description and the measured data to APPD. In a live environment the load coming into the system cannot be controlled .The new approach for the APPD is therefore limited by the incoming system load on which experiment plans we can use to analyze for performance problems. That means the new approach can only detect the performance problems that can be identified by the applicable experiment plans.

3.5 Reporting Issues

3.5.1 Reporting to Gropius

We use Gropius as the issue management system of choice, because it is ideally suited to handle a component based architecture. In Gropius a microservice system is represented as a project. Each individual service of the microservice system is represented as a component of the project. To be able to report an issue with an individual service to the right place, the explanation system needs the Gropius component ID of that service. Using the Kubernetes API, the explanation system can auto discover every deployment in a cluster. Then the user can add the corresponding component ID for each deployment. Even though a project has also an ID, it is sufficient to know the component ID to report the issue to the right place.

3.5.2 Horizontal Pod Autoscaler Issues

When we use the Kubernetes API to describe a Horizontal Pod Autoscaler for example the following information is returned:

```
Name:                                               uibackend
Namespace:                                          default
Labels:                                             <none>
Annotations:                                        metric-config.external.traffic.prometheus/
query:
                                                   standings w
http_server_requests_seconds_count{application='uibackend'}[1m]))
                                                    metric-config.traffic.prometheus/interval: 15
s
```

```
CreationTimestamp:                                Sat, 11 Dec 2021 11:59:28 +0100
Reference:                                        Deployment/uibackend
Metrics:                                          ( current / target )
  "traffic" (target average value):      0 / 5

Min replicas:                                     1
Max replicas:                                     4
Deployment pods:                                  1 current / 1 desired
Conditions:
  Type            Status  Reason                  Message
  ----            ------  ------                  -------
  AbleToScale     True    SucceededGetScale       the HPA controller was able to get the
target's current scale
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a
replica count from the traffic metric
  ScalingLimited  False   DesiredWithinRange      the desired count is within the acceptable
range
```

We see the name of the Horizontal Pod Autoscaler under `Name`, the deployment it is scaling under `Reference` and the namespace under `Namespace`. Some parts of the configuration of the Horizontal Pod Autoscaler are displayed, the minimum and maximum replica sizes, which metrics are used for scaling and under `Annotations` the query used for the custom metric `traffic`. Under `Metrics` we can see the current and desired value of the metric and the current and desired number of pods under `Deployment pods`.

There are also three different status conditions the Horizontal Pod Autoscaler can have: `AbleToScale`, `ScalingActive` and `ScalingLimited`. They can be either `true` or `false`. A reason and message are provided for each status condition to explain the current state. In the following we describe what they are, what they indicate and what could be the cause a bad status condition.

AbleToScale

The first status condition is `AbleToScale`. If the status condition is `true` the Horizontal Pod Autoscaler can fetch and update the scales of deployments. If the status condition is `false`, the Horizontal Pod Autoscaler is not able to get or update the scale of a deployment. A `false` status condition mostly indicates that the resource that should be scaled is not available. This can have multiple reasons, such as a faulty configuration of the Horizontal Pod Autoscaler where the target name is not correct.

ScalingActive

The ScalingActive status condition indicates if the Horizontal Pod Autoscaler is enabled and is able to fetch the required metrics to calculate the desired replica size. If the status condition is false, the Horizontal Pod Autoscaler is either not enabled or cannot fetch the metrics. This can have several causes. If the desired replica size was manually set to zero, then the Horizontal Pod Autoscaler is disabled. To detect this problem, the explanation system needs to determine if the replica size in the resource description is currently zero.

Another cause could be the three different metric APIs of Kubernetes. They could be either not enabled at all, or the implementations of the API could have crashed.

Lastly, there could also be a problem with the configuration of either the Horizontal Pod Autoscaler or the Metric Server. For example, the name of the metric in the configuration could be false. If a Prometheus query is used for custom metrics, and the query returns `null`, then the scale can not be properly calculated. For example, if we have a metric that counts the numbers of HTTP requests with status code `500` and there are currently no failed HTTP requests then when queried the metric returns `null` instead of zero.

ScalingLimited

If the status condition ScalingLimited is false, the needed replica count is within the configured boundaries of the Horizontal Pod Autoscaler. For instance if the Horizontal Pod Autoscaler is configured to scale between 2 and 5 pods and the current needed replica size is 3, then the ScalingLimited status condition is false. If the ScalingLimited status condition is true, then the needed replica size is either higher or lower than the configured boundaries. Which of these two cases is currently happening is described by the reasons `TooFewReplicas` or `TooManyReplicas`.

Catching Bad Status Conditions

Catching a bad status condition requires recurring monitoring of the status conditions. This is needed because only if metrics are not available for the Horizontal Pod Autoscaler an event is emitted in Kubernetes. This means only for the `ScalingActive` status condition a reactive approach would be feasible.

Different approaches can be used for the recurring monitoring. The status conditions could always be checked if a certain time period has passed since the last scaling event. To capture a `true ScalingLimited` closer to the time of occurrence, the status condition should be checked more often when a deployment is currently on the minimum or maximum replica size.

Reporting a Bad Status Condition

Once a bad status condition is detected we can report it to the Gropius backend using the GraphQL API. The issue includes relevant data and a text block consisting of the data and some predefined parts. The text blocks vary depending on the status condition responsible for triggering the issue creation. The body of the issue contains the following information:

- Name of the Horizontal Pod Autoscaler
- The component scaled by the Horizontal Pod Autoscaler
- The namespace
- Minimum and maximum replica size
- The current and desired pods

- The status condition with status, reason and message

In addition to the this information, every issue gets a custom textual explanation. The text of the explanation depends on the status condition and the reason given. In the following the textual explanations for each bad status condition are listed:

AbleToScale: `false` Reason: FailedGetScale:

```
The Horizontal Pod Autoscaler was not able to get or update the scale of the
<deployment name> deployment.
The Horizontal Pod Autoscaler gave the following reason for the problem: <message>.
Please check the attached details for further information.
```

ScalingActive: `false` Reason: FailedGetResourceMetric:

```
The Horizontal Pod Autoscaler was not able to fetch or calculate the metrics for the
<deployment name> deployment.
The Horizontal Pod Autoscaler gave the following reason for the problem: <message>.
Please check the attached details for further information.
```

ScalingLimited: `true` Reason: TooFewReplicas:

```
The <deployment name> has reached its maximum replica size of <max replica size>.
The <metric name> still exceeds the desired value of <desired value>.
To achieve the desired metric value an actual replica size of <needed replica size> would be required.
```

ScalingLimited: `true` Reason: TooManyReplicas:

```
The <deployment name> has reached its minimum replica size of <minimum replica size>.
The <metric name> is still below the desired value of <desired value>.
To achieve the desired metric value an actual replica size of <needed replica size> would be required.
```

3.5.3 Performance Problem Found

The problem is reported to the issue management system, after a performance problem is found. The following information is added to the issue:

- Name of the performance problem
- Where the performance problem occurs in the microservice system
- The classification of the system load
- The experiment plan used for finding the problem
- The metrics monitored during which the problem was found

In addition a small textual explanation, like the following, is presented at the start of the issue.

```
A performance problem has been found. The type of the performance problem is <problem type>.
The problem occurs in the following component: <place of problem occurrence>. Please check
the data provided in the issue for additional information.
```

4 Implementation

In this chapter, we describe how we set up and configured the test system. Then we show the parts of the concept that where implemented. The complete implementation is shown in Figure 4.1.

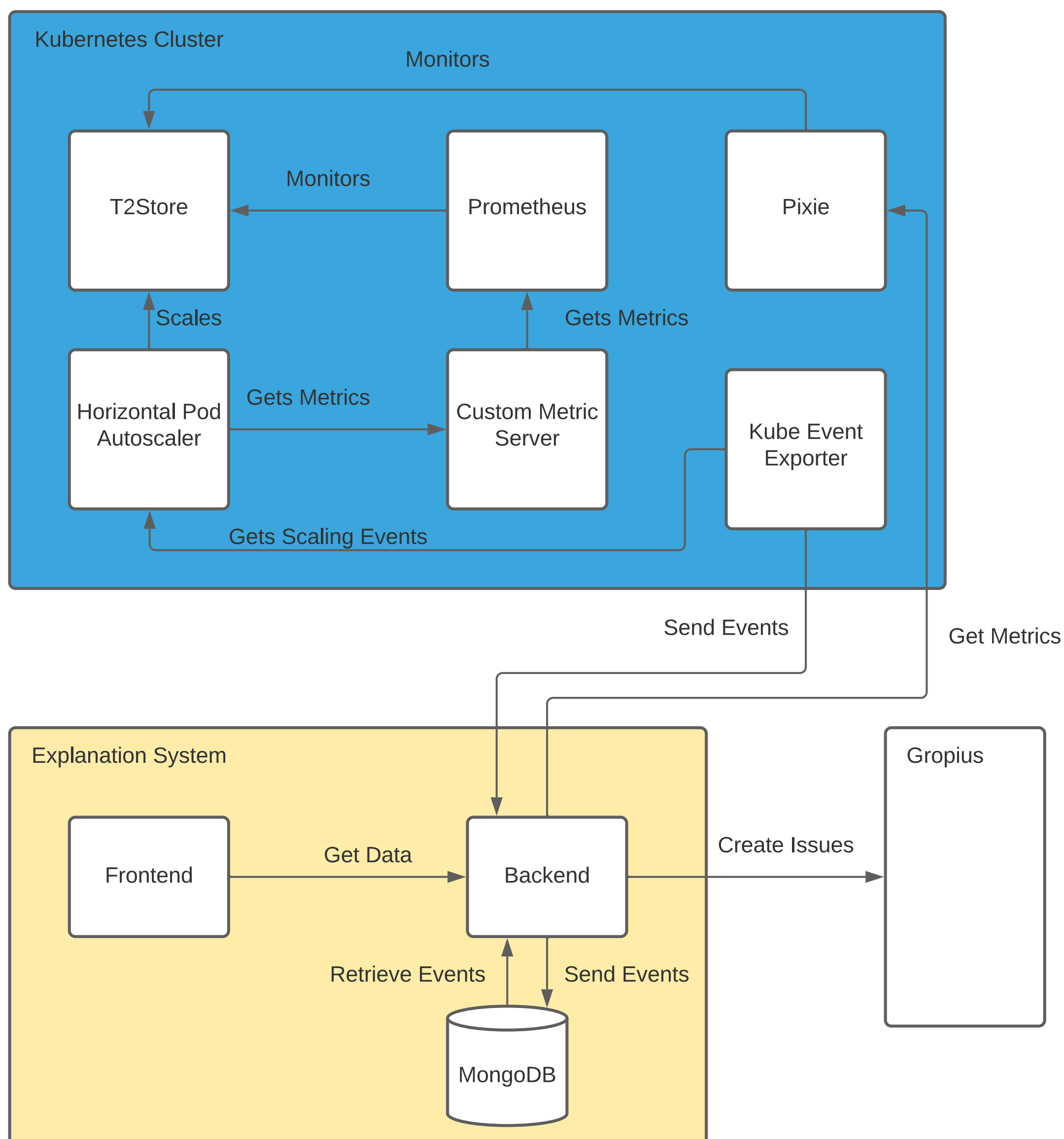

Figure 4.1: Implemented system.

4.1 Implementing the Test System

Minikube

We used Minikube to set up the Kubernetes cluster for the test system. A default Minikube cluster uses 2 CPU cores, 2GB of memory and 20GB of disk space. During testing with the default Minikube cluster and the T2Store we saw that one service of the T2Store already consumes about 200-350 MB of memory and up to 0.35 CPU cores under load. The T2Store consists of 8 services giving us a baseline of around 2GB memory and 2 CPU cores needed for operation without scaling. When a service first is starting up in Kubernetes the memory consumption is usually a bit higher than when the service is operational. Now if we want to scale the T2Store by a maximum of three replicas we need to provide Minikube with around a triple of the calculated resource baseline. To have some additional room we provided Minikube with 8GB of memory and 6 CPU cores. The Minikube cluster was set up with the following command:

```
minikube start --memory 8192 --cpus 6
```

The cluster was running on the following hardware:

- CPU: AMD Ryzen 5 5600X
- Memory: 16GB DDR4 3200MHz

Deploying the T2Store

After setting up the Kubernetes cluster with Minikube we deployed the T2Store. First, the dependencies of the T2Store were installed. Kafka and a MongoDB were installed using Helm charts. Then the deployment and service YAML files that the T2Store provides for Kubernetes were applied. This deployed a total of 8 services to the Kubernetes cluster. Then Prometheus was installed using the kube-prometheus-stack Helm chart.

During the development of the explanation system, I noticed an issue with the configuration of the T2Store deployment files. The image pull policy was set to `always`. This will result in all new images being pulled from the remote repository each time the Kubernetes cluster is started. Since the Minikube cluster was running on a local developer machine, the cluster was started and shut down each day. Because the T2Store was still in development the images on Docker Hub as well as the name of the images were regularly changed. This lead to two main problems:

The first problem is the available disk space for the cluster. Each older image is cached on the disk space of the cluster. This together with the regular updating of the images of the explanation system during its development led to a full disk for the cluster. The full disk caused that images could not be pulled and some services could not be started on the cluster. To find the cause of the problem a lengthy debugging session was needed. In the end, I localized the problem by directly connecting to the Minikube cluster using the command `minikube ssh`. Inside the Minikube cluster, I could use the `df -h` command to see that the main file system was full. To fix that problem we enabled the Kubernetes garbage collection. It automatically deletes unused images once the usage of the disk space approaches 80 percent.

The second problem is the changing of the image names. Each time the naming schema of the images were changed, the images with the old name were deleted. This lead to images not being found on the start of the cluster and the services not being able to start. After some debugging, I changed the image pull policy in the deployment files of the T2Store from `always` to `never`. Because the version of the T2Store at the time of this change was sufficient for the testing and development of the explanation system, no new version of the T2Store was needed.

4.1.1 Horizontal Pod Autoscaler

When we first set up the Horizontal Pod Autoscaler we activated the Kubernetes Metric Server to be able to scale the T2Store based on the CPU utilization and the memory consumption. Since the T2Store is already instrumented with Micrometer we also wanted to use the metrics that Micrometer provides for scaling. This allows us to use a wider range of metrics for scaling. To use custom metrics for scaling with the Horizontal Pod Autoscaler we needed an implementation of the Custom Metrics API.

Since it is fairly easy to scrape the metrics that Micrometer exposes with Prometheus it is beneficial that an implementation can collect the metrics for scaling from Prometheus. Following the list of implementations of the Custom Metrics API in the Kubernetes Horizontal Pod Autoscaler documentation, I had a deeper look at two different implementations that both offered support for collecting metrics from Prometheus.

Prometheus Adapter

The first implementation of the Custom Metrics API that supports the collecting of metrics from Prometheus is the Prometheus Adapter[1]. The adapter gathers all metrics that are available to Prometheus. Only metrics that follow a defined form are exposed for the Horizontal Pod Autoscaler. This has a disadvantage for the use case of this thesis. The T2Store already uses Micrometer to expose metrics in a particular format. This format would need to be changed to adhere to the format required by the Prometheus Adapter.

Kube Metrics Adapter

The second implementation of the Custom Metrics API that supports the collecting of metrics is the Kube Metrics Adapter[2]. It provides collectors for different metric providers. This allows greater flexibility should the need for a different metric provider arise. The Prometheus collector follows a different approach than the Prometheus adapter. It allows using of custom queries. The metrics that are returned by the query are then available for scaling. Because of the support for custom queries, the option to add more metric collectors, and easier configuration the Kube Metrics Adapter was chosen for the test system.

[1] https://github.com/kubernetes-sigs/prometheus-adapter
[2] https://github.com/zalando-incubator/kube-metrics-adapter

Choosing the metrics

For choosing the Metrics to scale the T2Store with we had a look at the Four Golden Signals described in the Site Reliability Engineering Book by Google [BJPM16]. For each type of signal we decided on one metric to use for scaling the services of the T2Store. Each metric is calculated by a custom Prometheus query.

For latency, Micrometer exposes two important metrics. `http_server_requests_seconds_count` is the total amount of requests a service receives at a specific endpoint. `http_server_requests_seconds_sum` is the sum of the duration of all requests that a service receives at a specific endpoint. To calculate the latency of a service with these metrics the Prometheus `rate()` function needs to be used on them. This gives the per second rate of the metrics measured over a time period. The time period has to be configured in square brackets and automatically handles missed scrapes and other imperfections in the metrics. Now the Prometheus `sum()` function needs to be used to get the metrics for all endpoints of a service. In the end, the total duration of all requests needs to be divided through the number of requests to get the latency. This leads to the following query:

```
sum(rate(http_server_requests_seconds_sum{application="payment"}[1m]))
/sum(rate(http_server_requests_seconds_count{application="payment"}[1m])).
```

For traffic we can take the `http_server_requests_seconds_count` metric and apply first the `rate()` function to get the per second rate and then use the `sum()` function to sum up all endpoints of a service. This gives us the following query to get the incoming traffic for a service:

```
sum(rate(http_server_requests_seconds_count{application="payment"}[1m]))
```

For error again the `http_server_requests_seconds_count` metric can be used. Again the `rate()` and the `sum()` function have to be applied. Now the status of the request needs to be filtered to only consider requests with a status of `500`. This gives the per second rate of requests with status `500`, dividing that by the total amount of incoming requests gets the percentage of requests that fail. This results in the following query:

```
sum(rate(http_server_requests_seconds_count{application="payment", status='500'}[1m]))
\sum(rate(http_server_requests_seconds_count{application="payment"}[1m]))
```

For saturation we can use the `container_cpu_usage_seconds_total` metric that is provided by Kubernetes to get the CPU utilization of a service. Here we have to filter for the namespace that the service runs in and search for all pods that belong to the deployment that will be scaled. We have to apply the `rate()` function to get the per-second CPU usage and use the `sum()` function to sum up the CPU usage of all pods belonging to that deployment. With that we get the following query:

```
sum(rate(container_cpu_usage_seconds_total{namespace="default", pod=~"payment-.*"}[1m]))
```

4.1.2 Configuring the Horizontal Pod Autoscaler

To choose the desired values for each metric, load testing was done using the Apache JMeter with the `t2-store-random-infinite.jmx` load profile provided by the T2Store. The load profile randomly adds one to five products to the cart of the T2Store and confirms the order. This process is repeated infinitely. In Figure 4.2 a visualization of the `t2-store-random-infinite.jmx` load profile from the documentation of the T2Store is shown. The load profile allows specifying the number of users it

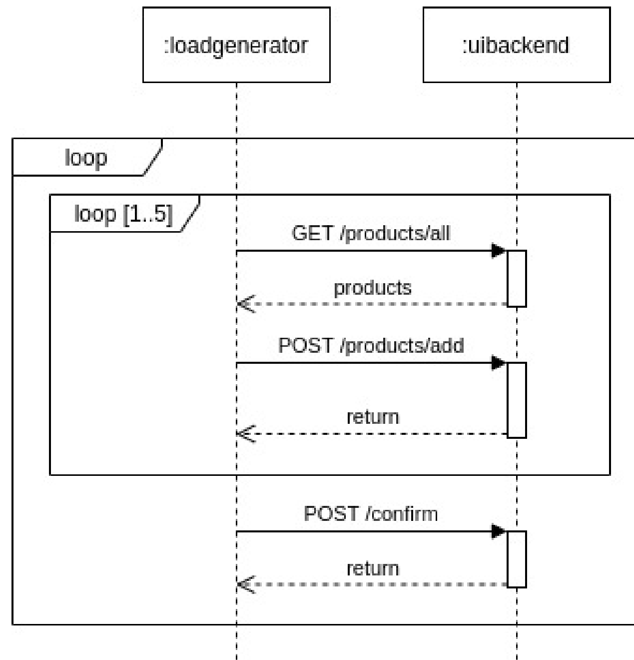

Figure 4.2: Visualization of the t2-store-random-infinite.jmx load profile[3]

should simulate and the time how fast users are added till the maximum amount is reached. This does not allow the creation of more complex load curves. To achieve more complex load curves the Ultimate Thread Group plugin for JMeter was installed using the JMeter Plugin Manager. With the Ultimate Thread Group plugin, it is possible to exactly specify a load curve. Using this the load curve in Figure 4.3 was created. It has different load behaviors like slow rise of load, sudden and short load spikes, and longer plateau phases where the load does not change. This allows testing the grouping methods of the prototype for different load behaviors. Using this improved load profile the following values were chosen for the four metrics:

- CPU: 150 millicores or 15% of a CPU core per pod
- Latency: 150ms response time
- Error: 10% error rate
- Traffic: 5 requests per second per pod

Using these values for the chosen metrics a configuration for the Horizontal Pod Autoscaler was created. In the annotations sections for each metric, the query interval and the PromQL query are configured. These annotations are needed for the Kube Metrics Adapter. It reads the queries from

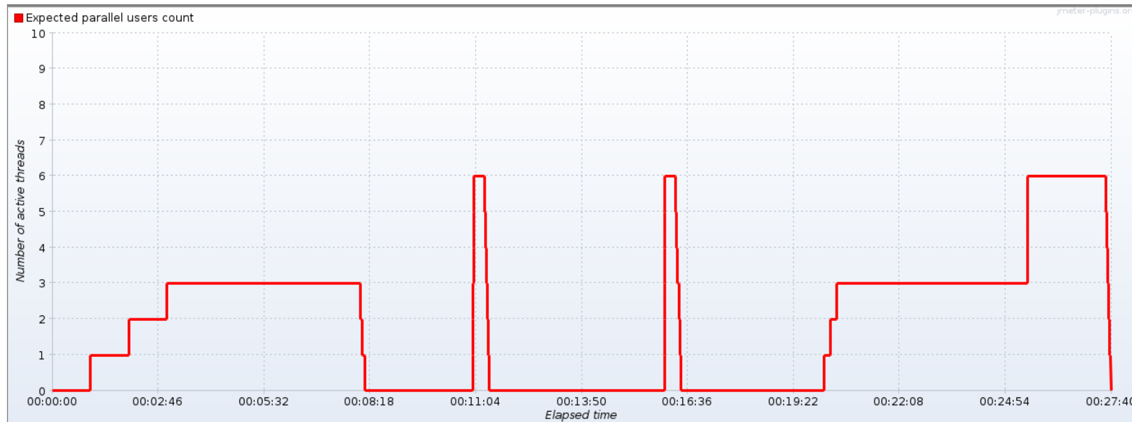

Figure 4.3: The new load curve.

the Horizontal Pod Autoscaler configuration, regularly queries the metrics, and then exposes them via the Custom Metrics API. Because the PostgreSQL database in the T2Store only supports a limited amount of connections and the strain on the test machine would be quite high with a large number of pods, the maximum replica size for each service was chosen to be four. The following is a sample configuration for the cart service of the T2Store:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: cart
  annotations:
    # This annotation is optional.
    # If specified, then this prometheus server is used,
    # instead of the prometheus server specified as the CLI argument `--prometheus-server`.
    # metric-config.<metricType>.<metricName>.<collectorType>/<configKey>
    metric-config.external.cpu.prometheus/query: |
      sum(rate(container_cpu_usage_seconds_total{namespace="default", pod=~"cart-.*"}[1m]))
    metric-config.cpu.prometheus/interval: "15s" # optional
    metric-config.external.error.prometheus/query: |
      rate(http_server_requests_seconds_count{application="cart", status='500'}[1m]) or on()
vector(0)
    metric-config.error.prometheus/interval: "15s" # optional
    metric-config.external.traffic.prometheus/query: |
      sum(rate( http_server_requests_seconds_count{application='cart'}[1m]))
    metric-config.traffic.prometheus/interval: "15s" # optional
    metric-config.external.latency.prometheus/query: |
      sum(rate(http_server_requests_seconds_sum{application="cart"}[1m]))/sum(rate(
http_server_requests_seconds_count{application="cart"}[1m]))
    metric-config.latency.prometheus/interval: "15s" # optional
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: cart
  minReplicas: 1
  maxReplicas: 4
```

```
metrics:
- type: External
  external:
    metric:
      name: cpu
      selector:
        matchLabels:
          type: prometheus
    target:
      type: AverageValue
      averageValue: "0.150"
- type: External
  external:
    metric:
      name: latency
      selector:
        matchLabels:
          type: prometheus
    target:
      type: AverageValue
      averageValue: "0.150"
- type: External
  external:
    metric:
      name: error
      selector:
        matchLabels:
          type: prometheus
    target:
      type: AverageValue
      averageValue: "0.100"
- type: External
  external:
    metric:
      name: traffic
      selector:
        matchLabels:
          type: prometheus
    target:
      type: AverageValue
      averageValue: "5"
```

Listing 4.1: HPA configuration for the cart service

4.2 Implementing the Prototype

4.2.1 Technology

The explanation system prototype is implemented as a frontend backend application. Both the frontend and the backend are written in Typescript[4]. Typescript is a superset of the JavaScript programming language. It adds type and type checking to JavaScript. This helps with developing and allows catching errors before runtime. When a TypeScript application is build the TypeScript compiler turns the TypeScript code into JavaScript code.

The frontend is implemented using the Angular framework[5]. Angular is based on TypeScript and offers Additionally we used Angular Material components[6] for building the UI.

The backend is implemented using the Nest.js framework[7]. Nest.js is a good pairing with Angular since they both use a similar way of structuring the application. This lessens the time needed to familiarize with both frameworks. For data storage, the backend is connected to a MongoDB[8]. MongoDB is a document oriented database that makes working with JSON documents very easy.

4.2.2 Exporting Kubernetes Events

To access the scaling events for Kubernetes the Kubernetes Event Exporter is used. It allows sending the scaling Events from Kubernetes to the prototype. The Event Exporter is configured to only send events from the Horizontal Pod Autoscaler to the prototype. These events include `SuccessfulRescale` event signifying that scaling happened as well as the can not get metrics events.

4.2.3 Extracting Data from Scaling Events

From the Kubernetes event the explanation system extracts relevant information. It uses the transformation process outlined in the concept chapter. Since scaling type, metric type, and replica size are in the message, a regular expression for each detail is used to extract the information. For the scaling type, two regular expressions one for scale out and one for scale in are needed. For the metric type, again need two regular expressions, one for internal and one for external metrics, are needed. Because the message for external metrics differs widely on the implementation of the Custom Metric Server a keyword for identifying the metric type needs to be used. This means that the name of the metric in the Horizontal Pod Autoscaler configuration needs to be embedded into the keywords configured for the explanation system. For example, if the custom metric `cpu` is used and the configured key word in the explanation system is `tomato`, the metric name in the Horizontal Pod Autoscaler configuration needs to be `tomatocputomato` so that the explanation system can detect the metric type. For extracting the replica size we use one regular expression.

[4]https://www.typescriptlang.org/
[5]https://angular.io/
[6]https://material.angular.io/
[7]https://nestjs.com/
[8]https://www.mongodb.com/

In the following we list the regular expressions used for extracting the information.

- Scaling out: `/(above)/`
- Scaling in: `/(below)/`
- External Metric: `/(?<=tomato)(.*?)(?=tomato)/`
- Internal Metric: `/reason: (.*) metrics above target/`
- Replica size: `/New size\: (\d+)/`

4.2.4 Grouping of Scaling Decisions

After the necessary information is extracted from the Kubernetes Event, they are evaluated for grouping. Groupings are based on the Kubernetes deployments. The current scaling event is always compared to the last scaling event from the same Kubernetes deployment. If there is no previous event in the database, a new set is created and the event is added to the set. If there are previous events the current event is compared to the one before.

Grouping by Scaling Type

First the scaling types are compared. If the scaling attribute of the current and the last event match, e.g. `'scaleOut' === 'scaleOut'`, the evaluation function returns `true` and the next evaluation function for grouping can be applied. If the scaling types do not match a new set is created and the event is added to the set.

Grouping by Time

After checking for scaling type, the evaluation function for grouping by time is called. Here the `createdAt` attributes are compared. If they are within a certain time the evaluation function returns `true` and the next evaluation function is called. If they are not within that time the events are not related, a new set is created and the current event is added to that set.

Grouping by Derivative

The last evaluation function that is called is the function for grouping by derivative. The current and the last event are compared by the metric used for scaling. A PromQL query gets the metric data between the `createdAt` time of the last and the current event. With a for loop, we subtract the n+1-th element from the n-th element of the metric data. If the result is positive the metric has increased and we add one to the `positivePercentage` variable. We repeat this /verb|n-1| times, with `n` being the length of the metric data array returned from the Prometheus query. After the for loop finishes, we divide the `positivePercentage` variable with the length of the metric data array. This gives us the percentage of the gradient of the metric curve that is positive. If a certain percentage of the gradient is positive the evaluation function returns `true`. If it does not return `true` a new set is created and the event is added to the set.

If all three evaluation functions return `true` the event is put into the same set as the previous related event.

4.2.5 Displaying the Events

The list of all events received by the explanation system is displayed using an Angular Material table. The table has five columns: The date and time of the event, the name of the deployment, the metric which was used for scaling, the scaling type, and the replica size. The table can be filtered for arbitrary values, all columns can be sorted and the table has a paginator.

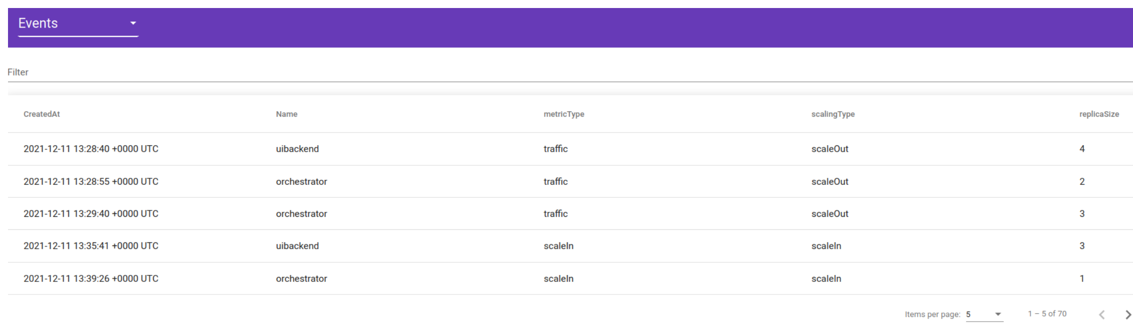

Figure 4.4: List of events.

4.2.6 Displaying the Sets of Events

Similarly to the list of all events, an Angular Material table is used to display the sets of events. The table has six columns: Date and time of the first and the latest event in the set, the name of the deployment, the metric used for scaling, the scaling type and the total count of events in the set. The table can be filtered for arbitrary values, all columns can be sorted and the table has a paginator. Each set of events has a detailed view that can be accessed by pressing on the row of the set.

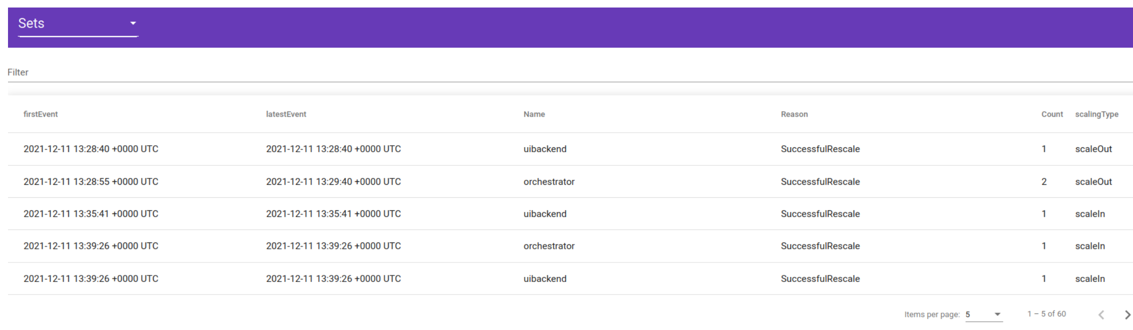

Figure 4.5: List of sets of events.

4.2.7 Mockups for the evaluation

In the following we show how we mocked some of the features described in the concept to evaluate the concept of the explanation system prototype.

Dashboard

To implement the dashboard to evaluate the concept, we used multiple different images of a service dependency graph of the T2Store. Each service was annotated with the current and maximum replica size as shown in the concept chapter. Several different versions of the service dependency graph with differing replica sizes were created. Additionally, we took images from the different metric graphs offered by Pixie in the Pixie UI. For the dashboard, we used images of the CPU, memory, and I/O usage. For the time slider, we used the Angular Material slider. The slider is hooked up to the images of the service dependency graph. Each time the slider is moved a new image is presented. This helps to better understand the concept of the time slider mechanism during the evaluation.

Figure 4.6: Screenshot of the dashboard.

Adding Metric Queries

The metrics that are used for scaling are displayed on a separate page. The metric and the query can be added by using two input fields and then saving the input with a button. A basic Angular Material table is used for displaying the metrics with the corresponding query. The metrics and their queries are retrieved from the database.

Metrics

Metric name

Metric query

Add metric

Metric name	Metric query
cpu	sum(rate(container_cpu_usage_seconds_total{namespace="default", pod=~"$deployment$-.*"}[1m]))
latency	sum(rate(http_server_requests_seconds_sum{application="$deployment$"}[1m]))/sum(rate(http_server_requests_seconds_count{application="$deployment$"}[1m]))
traffic	sum(rate(http_server_requests_seconds_count{application='$deployment$'}[1m]))
error	rate(http_server_requests_seconds_count{application="$deployment$", status='500'}[1m]) or on() vector(0)

Figure 4.7: Metrics query page.

Adding Gropius IDs

The Gropius IDs can be added on a separate page. The different deployments are displayed using an Angular Material table. Each deployment can be selected using a selector in the first column of the table. When the Edit ID button is pressed and a deployment is selected a dialog opens where the Gropius Project ID and the Gropius Component ID can be added.

Deployments

☐	Deployment name	GropiusProjectId	GropiusComponentId
☐	ui	5f574434fc13a003	5f574434fc13a005
☐	uibackend	5f574434fc13a003	5f574434fc13a007
☐	cart	5f574434fc13a003	5f574434fc13a009
☐	inventory	5f574434fc13a003	5f574434fc13a0a1
☐	orchestrator	5f574434fc13a003	5f574434fc13a0a3
☐	payment	5f574434fc13a003	5f574434fc13a0a5
☐	order	5f574434fc13a003	5f574434fc13a0a7
☐	creditinstitute	5f574434fc13a003	5f574434fc13a0a9

Edit IDs

Figure 4.8: Gropius ID page.

5 Evaluation

In this chapter, we discuss the process of evaluating the concept and prototype developed in this thesis. In section 5.1 it is described how the expert survey used for the evaluation was designed. In section 5.2 the results of the survey are shown and in section 5.3 the result of the evaluation are discussed. In section 5.4 the threats to the validity of the findings of this evaluation are addressed.

5.1 Goal Question Metric Approach

To evaluate the concept of this thesis an expert survey is held. The participants are shown a demonstration of the concept and the developed prototype followed by a survey using a questionnaire. The questions for the questionnaire were developed using the Goal Question Metrics approach [BCR94].

The first step of the Goal Question Metrics approach is identifying and defining one or more goals that capture what the thesis tries to achieve. The main goal of this thesis is to explore how the insights in the scaling behavior of a self-adaptive microservice system can be improved for developers and operators. Leading to the following formulation of the first and only goal:

G1 Improve the insight into the scaling behavior of a self-adapting microservice system for developers and operators.

The second step of the Goal Question Metric approach is identifying questions that characterize how the achievement of the goal can be assessed. A goal consists of three parts. These parts help find the questions needed to assess how well the goal has been achieved. The first part is the issue with the lack of good insights into the scaling behavior of a self-adaptive microservice system. Because of that, at least one question needs to aim at the issue that is currently present. The second part of the goal consists of the process that this thesis is trying to improve. The third part is the developer/operator that is the viewpoint from which the goal needs to be fulfilled for a successful thesis. Considering these three parts of the goal the following three questions were derived:

Q1 What problems do developers/operators face when trying to understand the scaling behavior of a self-adapting microservice system?

Q2 Which of the problems from **Q1** are solved by this thesis and to what degree?

Q3 Would developers/operators use the tool proposed in this thesis to solve the present issues?

The third step of the Goal Question Metric approach is finding appropriate metrics for each question. Since the plan for evaluating this thesis is to conduct an expert survey it needs to be analyzed if an expert survey is an appropriate metric for each of the questions. An expert survey is an appropriate metric for **Q1** if every expert has at least some experience working with self-adapting

microservice systems. Not every expert needs to have extensive experience working with self-adapting microservice systems. It is expected that the problems that developers/operators face differ depending on the amount of experience they have. It would therefore be beneficial if the amount of experience varies with the group of experts.

For **Q2** and **Q3** an expert survey is an appropriate metric if the experts are given an introduction to the concept developed in this thesis. Additionally, the experts need to satisfy the requirements of **Q1**.

Since an expert survey is an appropriate metric for each question, the only metric (**M1**) is an expert survey.

5.2 Expert Survey

5.2.1 Design and Realization

In preparation for the evaluation two presentations, one in German one in English, were created to help explain the problem are as well as the concept of this thesis. Both versions of the presentation are linked in the appendix. Additionally using the load profile described in section 4.1 scaling events were generated for the evaluation of the prototype. The load profile is also linked in the appendix.

Each expert survey was conducted via a Microsoft Teams meeting. First using the presentation in the preferred language of the expert the problem area of the thesis was explained. Then the expert was given a short overview of the concept. After we presented the concept of the thesis to the expert, the expert was presented with a questionnaire to assess the concept. The following questionnaire was created based on the questions **Q1** to **Q3** derived from the Goal Question Metric approach:

1. Which are the biggest challenges developers face when trying to understand/verify the autoscaling behavior of a microservice system?
2. Please order the following points by how much they challenge developers when trying to understand/verify the autoscaling behavior of a microservice system:
    a) High effort to provide information for analysis
    b) Information is scattered through to many tools/dashboards
    c) There are to many autoscaling decisions taking place to keep an overview
3. Which of the problems from questions 1 and 2 are solved by the system proposed in this bachelor thesis?
4. Which of the problems from questions 1 and 2 are reduced by the system in this bachelor thesis? If yes, how much are they reduced? (1-5) (1, not very much, 5, very much)
5. Are there features missing that you would need to better understand/verify the autoscaling behavior of a microservice system?
6. Would you use a completely implemented version of the tool proposed in this bachelor thesis?

Questions 1 and 2 aim to cover **Q1**. Question 2 was chosen to be able to get sufficient answers to Questions 3 and 4 even if the interviewed expert could not think of many answers to Question 1. Questions 3 and 4 aim to answer **Q2**. Question 5 is used to be able to improve the concept further or get an outlook on future work. The last question aims to answer **Q3**.

5.2.2 Results

Of the fifteen experts who agreed to take part in the evaluation twelve completed the review and survey. All of the experts are software developers with different levels of experience. Four of the experts work in the industry, two as full-time employees, two as working students. The rest of the experts are from academia. The companies range from a well-known global cooperation, to a IT consultancy and two smaller IT companies. The experience of the experts ranges between a few months and several years.

Question one yielded a wide range of problems the experts currently have with understanding the autoscaling behavior of a self-adaptive microservice system. The most mentioned problem was the poor presentation of the relevant data. Most of the developers mentioned primarily using the Kubernetes CLI and the Kubernetes Dashboard. This also relates to two other problems that were mentioned often. Kubernetes, autoscaling, and self-adaptive microservice systems have a steep learning curve and the huge amount of events and logs need extended expertise to find the relevant information. Several experts mentioned a lack of insight into the state of the microservice system in regards to autoscaling. One developer mentioned it is hard to get the right configuration for the autoscaling in the first place. Additionally, if a configuration is used it is hard to say if the configuration is sufficient or needs improvement. Three problems that were mentioned by three different experts are closely related. All three miss explanations of causal relationships within the self-adapting microservice system. One expert misses the relation between the metrics and how the autoscaling is taking place. While the second expert wants to see causal chains. This means how the scaling of one service has an impact on the rest of the system. While the third expert has trouble identifying if a violation of service levels through an increase or decrease in a metric is fixed by the autoscaling or changes in system load. Lastly, one expert mentioned the lack of metrics readily available. He mentioned that you either need to know all commands to access the data through the CLI or have to rely on external tools.

In the second question the experts were asked to sort three problems, that a developer/operator faces when trying to understand the autoscaling behavior of a self-adapting microservice system, by the impact on the developer/operator from highest to lowest. The problem with the highest impact was put in the first place while the problem with the lowest problem was placed last. Both problems **a)** and **b)** have a mean value of the placing of 1.8. With the highest placing of one and the lowest placing of three. The median placement is 1.5 for problem **a)** and 2 for problem **b)**. Interestingly the placement of problem **a)** was highly dependent on the experience of the interviewed expert. Experts with a lower amount of experience placed problem **a)** higher and the three experts with the most experience placed problem **a)** as the lowest impact problem. Problem **c)** was the lowest impact problem on average with an average placing of 2.1 and a median placing of 2.5. Problem **c)** was placed only once as the highest impact problem, but five times as the lowest impact problem.

The problems where questions 3 and 4 were answered together are summarized in this paragraph. The remaining problems where only question 3 was answered will be discussed afterward. First, the answers for the provided problems from Question 2 are shown, then for the problems mentioned in question 1. For the first 3 problems, all experts gave a rating. Problems four and five were rated by at least 4 experts. As can be seen by figures 5.4 and 5.5 the problem of visualization is solved very well by this thesis. All experts greatly appreciated the concept of the dashboard. They especially liked how the service dependency gives a temporal link between the scaling and the metrics of the system. Figure 5.1 shows that the concept of this thesis help provide relevant data for analysis. Experts also appreciated that the information available over the autoscaling behavior of a self-adaptive microservice system is more centralized as seen in figure 5.2. As shown in figure 5.3 the experts also appreciated the feature of grouping related scaling events into sets reducing the overall amount of events a developer needs to sort through.

Problem 1: High effort to provide information for analysis.

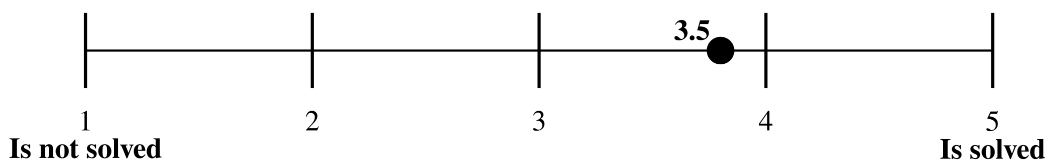

Figure 5.1: Average rating for **Problem 1**.

Problem 2: Information is scattered through to many tools/dashboards:

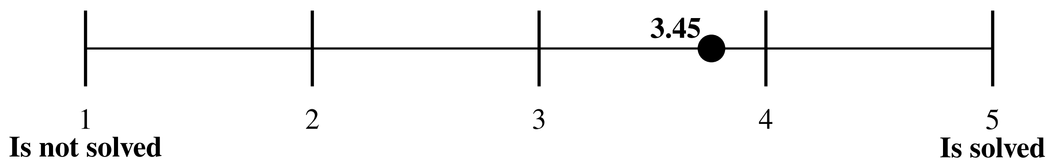

Figure 5.2: Average rating for **Problem 2**.

Problem 3: There are to many autoscaling decisions taking place to keep an overview:

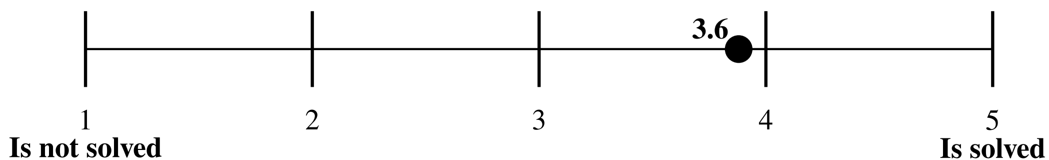

Figure 5.3: Average rating for **Problem 3**.

Problem 4: Poor visualization of data.

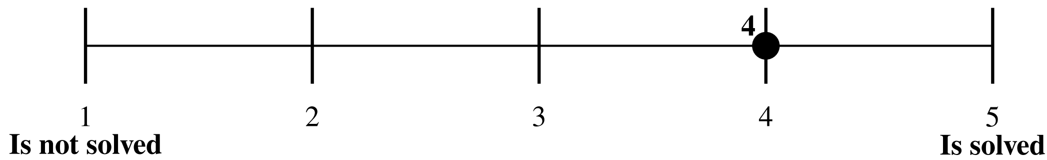

Figure 5.4: Average rating for **Problem 4**.

Problem 5: Lack of insight into the system state in regards to scaling.

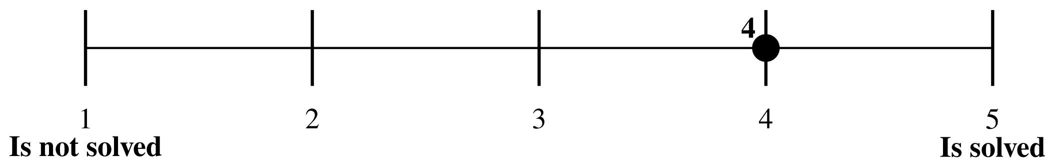

Figure 5.5: Average rating for **Problem 5**.

One expert noted that the provided concept can help newer developers themselves with familiarizing with autoscaling, reducing the learning curve. Another expert mentioned that the concept helps provide relevant data out of the box.

In question 5 the experts were asked for missing features and improvements. Most experts mentioned small UI improvements to the prototype. The most requested change was displaying not only the new replica size after a scaling event but also how much it has changed. Multiple experts wanted a tooltip to appear when hovering over the individual services in the service dependency graph. One expert also wanted a link to the rules by which a service is scaled. Another expert mentioned that the timeline of the service dependency graph could be annotated with markers that show exactly when events happen. Regarding the visualization of the events and the set of events in a list view, several experts mentioned that for large amounts of events a different way of displaying them might be needed. Several of the more experienced experts wished for a deeper analysis of the root cause of a scaling event. One expert wished to see causal chains of events. Which means how one event influences following events. Another expert wanted interactive explanations that depend on the needs of the recipient. It was mentioned that a deeper analysis could be provided by using a model that shows the expected behavior of the microservice system. Using that model differences between the actual and expected behavior of the system can be detected and reported.

After being asked question 6 all experts agreed that they would use a fully implemented version of the prototype, that was shown to them.

5.3 Discussion

In this section three hypotheses are created based on the questions **Q1** to **Q3** to evaluate the reaching of the goal **G1**. Each hypothesis corresponds to one question. If all hypotheses are accepted the goal **G1** is reached and this thesis is successful.

The following three hypotheses were created:

H1 Developers/operators face problems when trying to understand the autoscaling behavior of self-adapting microservice systems.

H2 The concept presented in this thesis solves some of these problems to some degree.

H3 Developers/operators can imagine using a fully implemented version of this concept.

The answers to question one in the survey show that developers/operators face a multitude of problems when trying to understand the autoscaling behavior of self-adapting microservice systems. As seen in question 2 the severity of some problems can depend on the amount of experience a developer has. Less experienced developers struggle more with getting enough relevant data in the first place. More experienced developers have an easier time getting the information, but they still need ways to better visualize and analyze the data. As developers face problems understanding the autoscaling behavior of self-adapting microservice systems regardless of their experience level the hypothesis **H1** is accepted.

Answers to questions 3 and 4 show, that the developers appreciate the concept presented to them. They especially liked how the concept offers more insight into the system state and offers better visualization for relevant data. Therefore hypothesis **H2** is accepted.

All experts agreed that they would use a fully implemented version of the concept that was presented to them. Therefore, hypothesis **H3** is accepted.

As all three hypotheses have been accepted, it can be concluded that goal **G1** has been reached and that this thesis is successful.

5.4 Threats to Validity

Runeson et al. [RH09] distinguish between four different aspects of validity: construct, internal and external validity as well as reliability.

Starting with the aspect of construct validity. It needs to be examined if the collected results correspond to the research question of this thesis [RH09]. There are some threats to validity in this regard because of the expert survey. The concept and prototype were only shown to the experts and they were not able to test the concept by themself, therefore it is possible that they either did not understand the concept well enough or needed a hands-on approach to get a better feel for the concept. This is somewhat minimized by the fact that the experts could ask questions to clarify unclear aspects of the concept. Another threat is the misunderstanding of the questions in the questionnaire. This is also alleviated by the fact that the interview was conducted in person and the experts could ask questions to clarify if there were uncertainties or ambiguities.

The second aspect is internal validity. It needs to be examined if the results gathered in this evaluation are caused by the concept of this thesis or if they are influenced by another source [RH09]. Because the experts were directly asked if the problems stated in the evaluation are solved or reduced by the concept of this thesis, no threat to the internal validity has been identified.

External validity is concerned with the question of whether the results of the evaluation can be generalized beyond the small sample size of the evaluation [RH09]. The sample taken in this evaluation was small compared to the population and the sample was also not selected at random. Only participants that the supervisor of this work already knew were contacted. The sample of this evaluation is therefore not very representative posing a threat to the external validity.

The last aspect is reliability. It is of concern whether the results of this evaluation can be replicated by other researchers [RH09]. The process of the evaluation and how the data for the evaluation was generated is described in section 5.2.1. and the presentation used for the evaluation can be found in the appendix. Therefore no major threat to the validity was identified. A minor threat is that the artifacts of this thesis are hosted on the internet and might not be available forever.

6 Conclusion

In this chapter the thesis is concluded with a summary in section 6.1. Then, we discuss some of the limitations of this thesis in section 6.2. Finally in section 6.3 we give an outlook into future work, that can be done using this thesis as a basis.

6.1 Summary

This thesis provides a concept that improves the visualization of the scaling behavior of a self-adapting microservice system. The concept also reduces the amount of information presented to the user. Additionally the concept offers a way of recognizing performance problems. Using the concept a prototype was implemented. The code of the prototype is available on Github[1].

The prototype was used for a better evaluation of the concept of this thesis. A expert survey was conducted. The evaluation shows that there are several problems developers face when trying to understand the autoscaling behavior of self-adaptive microservice system. The experts stated that the presented concept and prototype contributes to solving theses problems. Especially developers and DevOps engineers with less experience profit from this concept, because the concept lowers the rather steep learning curve. Though more experienced developers and DevOps engineers also profit from the better overview this concept offers for the autoscaling behavior of self-adaptive microservice systems.

6.2 Limitations

This thesis only looks at reactive autoscaling with the Kubernetes Horizontal Pod Autoscaler. It is not clear if the concept of this thesis is applicable to other autoscaling solutions. This highly depends on how other autoscaling solutions expose their inner workings.

The experts interviewed in the evaluation of this thesis are not representative for the complete population. It is therefore not possible to say whether the results of the evaluation of this thesis are transferable to the entire population.

[1] https://github.com/tobiasrodestock/rodestock_ba

6.3 Future Work

As seen in the Limitations section, the Horizontal Pod Autoscaler is the only autoscaler currently supported. Depending on how other autoscaler like the Amazon EC2 Auto Scaling solution expose scaling events, support could be added for them.

During the evaluation several improvements to the UI of the concept where proposed. In a future work these improvements need to be evaluated if they indeed improve the concept. Then the useful improvements could be added to the concept.

Another aspect of future work is a behavioral model of the monitored system. In the behavioral model is the expected behavior of the monitored system encoded. This would make it possible to detect differences between expected and actual behavior. Detecting unexpected behavior enables faster reporting of issues.

As Speth noted in [Spe21] frequently occurring violations of Service Level Objectives in self-adaptive microservice systems need to be reported as cross-component issues. These frequently recurring violations also need to be explained and are therefore another point of future work.

Using the information already available through the service dependency graph additional analysis could be done to examine how dependent services are affected by scaling of other services. Likewise there is the need to explore how the scaling of a particular service is impacted by the performance of other services.

Finally for explaining the behavior of self-adapting microservice systems, more work needs to be put into recognizing if autoscaling is successful in keeping the system in a healthy state. To enable that a way needs to be found to identify when a autoscaling event is successful with its scaling behavior.

Bibliography

[BCR94] V. R. Basili, G. Caldiera, H. D. Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 35).

[BGG+19] M. Blumreiter, J. Greenyer, F. J. C. Garcia, V. Klös, M. Schwammberger, C. Sommer, A. Vogelsang, A. Wortmann. *Towards Self-Explainable Cyber-Physical Systems*. 2019. arXiv: 1908.04698 [cs.AI] (cit. on p. 8).

[BJPM16] B. Beyer, C. Jones, J. Petoff, N. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, 2016. ISBN: 9781491929124. URL: https://books.google.de/books?id=81UrjwEACAAJ (cit. on p. 26).

[BK19] D. Bohlender, M. A. Köhl. "Towards a Characterization of Explainable Systems". In: *CoRR* abs/1902.03096 (2019). arXiv: 1902.03096. URL: http://arxiv.org/abs/1902.03096 (cit. on p. 4).

[Fow] M. Fowler. *Microservices Guide*. URL: https://martinfowler.com/microservices/ (cit. on p. 4).

[IBM05] *An Architectural Blueprint for Autonomic Computing*. Tech. rep. IBM, June 2005 (cit. on p. 3).

[KC03] J. Kephart, D. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055 (cit. on p. 7).

[KGG18] V. Klös, T. Göthel, S. Glesner. "Comprehensible and dependable self-learning self-adaptive systems". In: *Journal of Systems Architecture* 85-86 (2018), pp. 28–42. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2018.03.004. URL: https://www.sciencedirect.com/science/article/pii/S1383762117304472 (cit. on p. 8).

[Klö19] V. Klös. "Explainable Self-Learning Self-Adaptive Systems". In: *Explainable Software for Cyber-Physical Systems (ES4CPS)* (2019), p. 46 (cit. on p. 8).

[LR04] R. Laddaga, P. Robertson. "Self adaptive software: A position paper". In: (Jan. 2004) (cit. on p. 3).

[New21] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. English. Paperback. O'Reilly Media, Sept. 28, 2021, p. 616. ISBN: 978-1492034025 (cit. on pp. 3, 4).

[RH09] P. Runeson, M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empir. Softw. Eng.* 14.2 (2009), pp. 131–164 (cit. on pp. 40, 41).

[Ric18] C. Richardson. *Microservices Patterns Video Edition*. City: Manning Publications, 2018. ISBN: 9781617294549 (cit. on p. 6).

[SBB20] S. Speth, U. Breitenbücher, S. Becker. “Gropius—A Tool for Managing Cross-component Issues”. In: *Communications in Computer and Information Science*. Ed. by H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Camporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, U. Zdun. Vol. 1269. Springer. Springer, 2020, pp. 82–94 (cit. on pp. 6, 11).

[SBB21] S. Speth, S. Becker, U. Breitenbücher. “Cross-Component Issue Metamodel and Modelling Language”. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. INSTICC. SciTePress, May 2021, pp. 304–311. ISBN: 978-989-758-510-4. DOI: 10.5220/0010497703040311. URL: https://www.scitepress.org/PublicationsDetail.aspx?ID=Vtgn8x557mU=&t=1 (cit. on pp. 6, 11).

[SKBB21] S. Speth, N. Krieger, U. Breitenbücher, S. Becker. “Gropius-VSC: IDE Support for Cross-Component Issue Management”. In: *Companion Proceedings of the 15th European Conference on Software Architecture* (CEUR Workshop Proceedings). Ed. by R. Heinrich, R. Mirrandola, D. Weyns. CEUR, Oct. 2021. URL: http://ceur-ws.org/Vol-2978/tool-paper103.pdf (cit. on p. 7).

[Spe19] S. Speth. “Issue management for multi-project, multi-team microservice architectures”. MA thesis. 2019 (cit. on p. 6).

[Spe21] S. Speth. “Semi-automated Cross-Component Issue Management and Impact Analysis”. In: *Proceedings of 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. IEEE, 2021, pp. 1090–1094. DOI: 10.1109/ASE51524.2021.9678830. URL: https://ieeexplore.ieee.org/abstract/document/9678830 (cit. on p. 44).

[TVP+21] A. Tsagkaropoulos, Y. Verginadis, N. Papageorgiou, F. Paraskevopoulos, D. Apostolou, G. Mentzas. “Severity: a QoS-aware approach to cloud application elasticity”. In: *Journal of Cloud Computing* 10.1 (Aug. 2021). DOI: 10.1186/s13677-021-00255-5. URL: https://doi.org/10.1186/s13677-021-00255-5 (cit. on p. 8).

[Wer15] A. Wert. “Performance Problem Diagnostics by Systematic Experimentation”. PhD thesis. 2015. DOI: 10.5445/IR/1000048516 (cit. on p. 17).

All links were last followed on January 26, 2022.

A Artifacts of the Expert Review

For the expert review the following artifacts were used:

A.1 English Presentation and Questionnaire

The English version of the presentation and questionnaire used in the evaluation has been uploaded to: https://github.com/tobiasrodestock/rodestock_ba/tree/master/presentation/english .

A.2 German Presentation and Questionnaire

The German version of the presentation and questionnaire used in the evaluation has been uploaded to: https://github.com/tobiasrodestock/rodestock_ba/tree/master/presentation/german .

A.3 Load Profile

A custom load profile as shown in Section 4.1 was used to generate scaling events for the evaluation. The load profile has been uploaded to: https://github.com/tobiasrodestock/rodestock_ba/tree/master/presentation/load_profile .

A.4 Questionnaire Results

The answers to the questionnaire and the feedback gathered for the concept were written down in a PDF file. The raw notes are uploaded to the following place: https://github.com/tobiasrodestock/rodestock_ba/tree/master/presentation/results .

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.

Nufringen 26.01.2022 Radestock

place, date, signature