Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Evaluating of Feasibility and Aiding Explainability of Scaling Policies Using Architectural-based Simulations

Philipp Gruber

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Floriment Klinaku M.Sc.
Commenced:	May 10, 2021
Completed:	December 10, 2021

Abstract

Online services have become an indispensable part of our lives. The Internet of Things (connecting electronic devices to the Internet) has added another possible application in recent years. The resulting increase or fluctuation in the number of users means that these services must remain available and accessible. Disruptions and outages have serious consequences. If, for example, the scaling of a service does not function properly, the provider may incur considerable costs.

Cloud engineers work with so-called scaling policies to enable elastic systems that automatically scale resources above a certain threshold of a metric. To avoid mistakes, architecture-based simulations like Palladio's can help. Palladio's effectiveness has been demonstrated in various scenarios such as software as a service. However, whether Palladio is also effective in the context of scaling policy has not been sufficiently investigated so far.

The goal of this work is to investigate the feasibility of scaling policy simulation using Palladio. Subjects of investigation are the accuracy of the simulations and whether the Palladio model helps with the comprehensibility of scaling policies.

To determine the accuracy, measured values are recorded during an experiment and later compared to the Palladio simulation results. A Kubernetes cloud system from the MoSaIC project is used as a reference system. The use case of the project is container ships sending data, with the number of ships increasing due to a new customer. To generate the load, the load testing software Gatling is used. The experiment is divided into two phases, a scaling experiment and an elasticity experiment. The former is used to quickly rank the MoSaIC Kubernetes system, which is a prerequisite for the design of the elasticity experiment. The design provides two load scenarios (low and medium). With these scenarios, two different scaling policy configurations, which differ in terms of the threshold value, are put under load in the experiment. These scenarios, the system, and the scaling policies were modeled with Palladio.

During the modeling process, we found that various factors made it difficult to model the experiment scenario and run the simulation. The corresponding deficiencies and knockout criteria and possible workarounds to circumvent the problems were documented. The scaling policies could not be simulated to the full extent. Therefore it was not possible to simulate them. However, we were able to show the potential of Palladio and that the Palladio model we used allows the tracking of how self-adaptations were performed. That theoretically improves the understandability of the scaling policies.

Future work can build on our findings and find out via further experimentation whether the documented deficiencies can be fixed or circumvented. In addition, an experiment should be conducted to investigate whether the improved understandability of the scaling policies through Palladio can also be proven.

Kurzfassung

Onlinedienste sind in unserem Leben nicht mehr wegzudenken. Mit dem sogenannten Internet der Dinge (Vernetzung von elektronischen Geräten mit dem Internet) kam in den letzten Jahren eine weitere Anwendungsmöglichkeit hinzu. Die daraus resultierende steigende beziehungsweise schwankende Anzahl an Nutzern setzt voraus, dass diese Dienste verfügbar und erreichbar bleiben. Störungen und Ausfälle haben gravierende Folgen. Sollte beispielsweise die Skalierung eines Dienstes nicht ordnungsgemäß funktionieren, können erhebliche Kosten auf den Anbieter zukommen.

Cloud-Ingenieure arbeiten mit sogenannten Scaling Policies, um elastische Systeme zu ermöglichen, die automatisch ab einem gewissen Schwellwert die Ressourcen skalieren. Um Fehler zu vermeiden, können Architektur-basierte Simulationen wie von Palladio helfen. Palladios Effektivität wurde in verschiedenen Szenarien wie zum Beispiel Software as a Service nachgewiesen. Allerdings wurde bislang nicht ausreichend untersucht, ob Palladio auch im Kontext Scaling Policy effektiv ist.

Das Ziel dieser Arbeit ist die Untersuchung der Machbarkeit von Scaling Policy Simulationen mit Palladio. Untersuchungsgegenstände sind die Genauigkeit der Simulationen und ob das Palladio-Modell bei der Verständlichkeit der Scaling Policies hilft.

Um die Genauigkeit zu bestimmen, werden Messwerte während eines Experiments aufgezeichnet und später mit den Ergebnissen der Palladio Simulation verglichen. Als Referenzsystem dient ein Kubernetes Cloudsystem des MoSaIC-Projekts. Der Anwendungsfall des Projekts sind Containerschiffe, die Daten senden, wobei die Anzahl der Schiffe durch einen neuen Kunden steigt. Um die Last zu erzeugen wird die Lasttestsoftware Gatling verwendet. Das Experiment ist in zwei Phasen unterteilt, in ein Skalierungs- und ein Elastizitätsexperiment. Ersteres dient zur schnellen Einstufung des MoSaIC-Kubernetessystems, welches eine Vorbedingung für den Entwurf des Elastizitätsexperiments ist. Der Entwurf sieht zwei Last-Szenarien (schwach und mittelstark) vor. Mit diesen Szenarien werden im Experiment zwei verschiedene Scaling Policy- Konfigurationen, welche sich hinsichtlich des Schwellwerts unterscheiden, unter Last gesetzt. Diese Szenarien wurden genauso wie das System und die Scaling Policies mit Palladio modelliert.

Während des Modellierungsprozesses haben wir festgestellt, dass diverse Faktoren es erschweren ein Modell für das von uns verwendete Experimentsszenario zu erstellen und die Simulation auszuführen. Die entsprechenden Mängel und K.-o.-Kriterien wurden dokumentiert, genauso wie mögliche Hilfskonstruktionen, um die Probleme zu umgehen. Wobei letztendlich die Scaling Policies nicht vollständig modelliert werden konnten und es daher nicht möglich war diese zu simulieren. Allerdings konnten wir das Potenzial von Palladio aufzeigen zeigen sowie, dass das verwendete Palladio-Model die Nachverfolgung, wie Selbstanpassungen vollzogen wurden, prinzipiell ermöglicht. Womit die Verständlichkeit der Scaling Policies theoretisch verbessert wird.

Zukünftige Arbeiten können auf unseren Erkenntnisse aufbauen und über weitere Experimente herausfinden, ob sich die dokumentierten Mängel beheben oder umgehen lassen. Zudem sollte mit einem Experiment untersucht werden, ob sich die verbesserte Verständlichkeit der Scaling Policies durch Palladio auch belegen lässt.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The popularity of IoT (Internet of Things) systems is growing because with the connection to the internet - trivial and well-known devices trigger a new user experience. Through this evolution, these things get smart, e.g., Smart Home. Taivalsaari and Mikkonen [TM18] put it: “enriching Thing X with an Internet connection, Services, and Apps - Thing X gets smart, or Smart Thing X”. There are hardly any limits to the intended use. Gubbi et al. [GBMP13] for example, list the seemingly infinite potential application areas.

Here, cloud computing becomes crucial because a Smart Thing X, as a vessel or any other device, sends data from its sensor to a server, allowing it to process, store and access the data for further use. That results in new challenges software developers and architects face. One major challenge is the adequate provisioning or the scaling of computing resources. These systems should not only be scalable but also be elastic. In cloud terminology elastic means a system is automatically self-adaptive. The main goal is to be as cost-efficient and available as possible. It means to prevent over-provisioning and under-provisioning by meeting the SLOs (Service level objectives) and not overspending for unused resources.

Doing that sounds relatively straightforward and trivial. However, a recent example from a German Covid vaccination registration website is not the only one where users still experience an unavailable service, and the providers miss their SLOs [1].

It does not matter what exactly triggered the unavailability in this case. Whether the lack of server capacity or a more complex architectural error that induced a bottleneck played are role. There are tools to prevent such outages and build robust elastic cloud services.

Palladio, for example, is a model-driven performance predicting simulation tool [RBH+16] that has proven to be effective for cloud use-cases, as Lehrig [Leh14], and Klinaku et al. [KBB19] showed. However, Klinaku et al. [KBB19] reported several shortcomings. Therefore, it remains unclear if Palladio would be ready to simulate scaling scenarios within a Kubernetes environment. In the case that Palladio can successfully simulate scaling policies, it is crucial to investigate how accurate the results are. It becomes more challenging for software architects to develop elastic services because the systems become more complex. For instance, think about a microservices system with different services, interfaces, and more than one possible use case. Here it would be costly to scale the whole system if only one or two services are the bottleneck. This issue creates the need for simulation tools to support developers. This thesis has its focus on feasibility and accuracy in that regard. Firstly, this thesis investigates the feasibility of modeling such a system. A possibility to realize such an autoscaling system is Kubernetes - a state-of-the-art technique. Therefore, a Kubernetes system is selected as the reference system. Additionally, we assessed the accuracy of the performance

[1] https://www.t-online.de/region/duesseldorf/news/id_89344254/startprobleme-bei-der-vergabe-von-impfterminen-an-aeltere.html

predictions and the ability to make decisions based on that. Furthermore, we examined whether such a model helps to foster the explainability of such a system and its scaling behavior. This thesis investigates the feasibility of simulating scaling policies with the architectural-based Palladio approach. We evaluate the accuracy of the created simulations and the explainability of the scaling policy model.

We measure the response time to determine the accuracy through an experiment on a Kubernetes cloud system from the MoSaIC project. Kubernetes is a state-of-the-art technique to realize cloud systems independent of the cloud provider. The use case of the project is vessels sending data. We design an experiment where the number of vessels increases due to a new customer. That increase should test our scaler and the scaling policy in place. Our scenario is to evaluate different scaling policies regarding their effectiveness, as in a realistic use case where a developer wants to opt for the optimal configuration.

The experiment is divided into two phases, a scaling, and an elasticity experiment. The scaleability phase benchmarks the Kubernetes system, and the elasticity experiment tests the auto scaler with the scaling policy. We use the load test software Gatling for the load generation and record the measurements. Later we compare the measurements with the results of the Palladio simulation. Lastly, we examined whether such a model helps to foster the explainability of such a system and its scaling behavior.

Thesis Structure

Here, give an overview of your thesis structure.

Chapter 2 – Foundations: Here, we provide the information about Kubernetes, the MoSaIC project, Cloud Computing, Scaling Policies, Architecture-based Simulations, Palladio, and the explainability. In short, everything to understand our research.

Chapter 3 – Related work: The related work is separated into the analysis of AutoScaleSim and analyses of Palladio's SimuLizar in the context could systems and scaling.

Chapter 4 – Research Design: Here, we describe our research questions, how we designed our research regarding the process, the experiment, and the methodology.

Chapter 5 – Results: The results chapter is separated into four parts, how and what scaling policy we choose, the conduction and results of our experiment, the Palladio model results, and the simulation results.

Chapter 6 – Evaluation: Here, we provide the evaluation of our research questions.

Chapter 7 – Conclusion: We conclude our thesis by summarizing the results, highlighting our contributions, describing the lessons learned, considering the limitations, and taking a look at possible future work.

2 Foundations

In this chapter we introduce the domain of this thesis, the used technology, and systems.

This thesis investigates the feasibility, accuracy, and explainability of predictions of scaling policies in a cloud environment. To introduce the reader to the topic, we explain cloud computing and scaling policies. Then we describe Kubernetes with which the demonstrator or experiment system got created. Next we introduce the demonstration system, which is part of the MoSaIC project. Further, we do the same with the domain architecture-based simulations and technology of the predictions - Palladio, which is used to model the experiment system. Lastly, we give an overview of what is explainability.

2.1 Cloud Computing

According to the NIST Definition of Cloud Computing [MG11], the domain of cloud computing describes accessing computing resources via the network. Of course, in a manner that these resources can be rapidly provisioned or released. Often users have no idea about the underlying layer of services that they use, but they expect that the service is available and reacts quickly. It is a black box to them, and they do not know that the computing resources run distributed in a cluster.

Cloud computing allows such a distributed cluster. In this cluster, services can be scaled when they are needed to cope with a higher load or server failure occurs. Ideally, this should work self-adaptive or automatically. Of course, administrators who observe such a system and react manually are more expensive, error-prone and unlikely as fast as the self-adaption mechanism. With these prospects, it makes sense to follow that route. Although, humans are still needed, for example, to assist or observe.

Scalability is not the property that guarantees a self-adaptive behavior, which avoids inefficient over-provisioning or a costly under-provisioning. It states the possibility that you can scale a system and that there are no architectural or technological hurdles that prevent that. In contrast, elasticity describes provisioning respectively deprovisioning resources automatically to an expected respectively experienced workload [HKR13]. The reason to take such a scaling decision is to avoid the unsound effects of an overload or an oversize of the available resource. In particular, to ensure that the used resources are used as efficiently as possible.

In a world where cost-efficiency is a competitive advantage over competitors, it is crucial to be as elastic as possible with the provided services. This is another motivation why this issue is relevant. Yet, achieving this elasticity efficiency is not trivial. We describe a method to that in the following section.

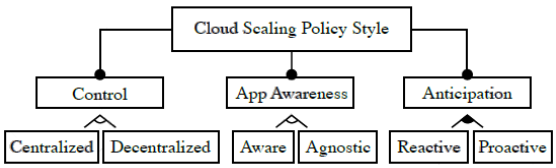

Figure 2.1: The different styles of scaling policies, from [KHB21].

2.1.1 Scaling policies

In some literature self-adaption rules is the name of a measure to realize elastic cloud systems. In others, it is called an auto-scaling algorithm. However, we think the more precise term is scaling policy. In our work, we use the term scaling policy, as this is state of the art. The cloud provider Amazon Web Services (AWS) defines the term scaling policy for EC2 (Amazon Elastic Compute Cloud) [1] as follows: "choose scaling metrics and threshold values for the CloudWatch alarms that invoke the scaling process." Apart from the used technology, a scaling policy manages the scaling of a system depending on the demand. The approach of the policy to observe the demand becomes decided through the selected metric. Klinaku et al. [KHB21] describes a scaling policy terminology, which is threefold: First, the if conditions for the action, and second, the actions to take. These actions would be the scaling strategy. And third, constraints under which the system should stay in a steady state. For instance, a simple example for a scaling policy that adds additional computing resources is the following:

Threshold: 90 % CPU utilization

Action: Add one add

Constraint: No further scaling action in the next minute

The AWS documentation recommends using central processing unit (CPU) utilization as a metric. To explain our example from above in more detail, we pick 90% CPU utilization as the threshold for increasing to an additional instance. The system has to follow the constraint to do no further action in the next minute. When we change the threshold from 90% to 85%, we change the configuration of the policy, according to Klinaku et al. [KHB21]. Apart from different policy configurations, they can differ in their style, too. In terms of style, a scaling policy can be decentralized or centralized. A decentral policy may replicate a specific service while leaving the whole system, which consists of more than a single service, untouched. On the contrary, a central policy scales the whole system and does not focus on a specific service or component of the system. In addition [KHB21] not only mention the control part as a style feature, but application awareness and anticipation are also factors. See Figure 2.1 for an overview of the scaling policy styles.

Application awareness means that the policy is either aware or agnostic. Aware means the policy is aware of services or applications that are running on the system. When it is agnostic, it does not take the architecture into account. An example of an agnostic policy is when the system scales based on utilization metrics like CPU/RAM (Random Access Memory) and does this without further

[1] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html

knowledge. In contrast, designing a scaling policy concerning the architecture of a system makes the policy style aware. That means using the length of a message queue to scale does imply awareness of the service that uses the message queue. However, if additional knowledge about a bottleneck component can be scaped through the message queue length to execute fine-grain scaling actions, then the policy can be considered aware [KHB21].

The anticipation distinguishes between reactive and proactive. That means either foreseeing an increasing load and scaling up proactively - or scaling up as a reaction to increasing load. It could be possible to have a hybrid approach that uses reactive and proactive elements.

As said in the beginning, the term policy is fuzzy like Han et al. [HGGG12] have discussed previously. They introduce scaling algorithms, where the main difference is that those algorithms act more fine-grained than just adding or removing VM instances. We do not differentiate between scaling algorithms and policies under the condition that this policy or algorithm consists at minimum of a trigger, action, and constraint. In addition, technology-specific mechanisms are not part of that [KHB21]. For that, we refer to the Kubernetes/MOSAIC section. In addition, there are several possibilities to scale, for instance, vertically or horizontally or in some hybrid form, as done by [BSM20]. To explain that, horizontal scaling means to in- or decrease the number of replicas. Vertical scaling denotes the scaling of the computing power of an instance. That is a design decision that is not limited by the technology, like Kubernetes, which is introduced in the next Section 2.2.

2.2 Kubernetes

Kubernetes is one possible technology to realize Cloud Computing. Moreover, it is an open-source system initially developed by Google (now maintained by the Cloud Native Computing Foundation CNCF) and a de facto standard to deploy containerized services in the cloud [NYK+20]. Kubernetes serves as an orchestrator for these containers. A (docker) container is a unit of software that packages code and its dependencies into a deployable unit [2].

According to Burns et al. [BBH18], with this container structure, Kubernetes has the following four main benefits:

- Velocity
- Scaling
- Abstraction of infrastructure
- Efficiency

The property of velocity does not only denote speed. It also refers to the ability to react quickly and still be available. In some regard, this overlaps with the scaling of the system. However, scaling means only adapting to a changing workload. It does not mean to ship updates iteratively. In a competitive situation and with continuous integration used, this is at least as crucial as scaling. Immutable containers are the approach to achieve that. Although it is technically possible to update an existing Docker container image, the core concept is to avoid it. Hence, containers get updated through replacement. This method is beneficial because the old container gets not overwritten and

[2] https://www.docker.com/resources/what-container

is still available for a potential rollback in case of an error. In addition, by applying infrastructure as code, with documenting the desired state of the system or container declarative in a configuration file, it is avoided that a wrong action leads to an incorrect state. Further, the desired state gets documented, and if a mismatch occurs self-healing countermeasures will be taken because the system monitors the status continuously.

There is another reason why scaling got mentioned separately. On the one hand, the term refers to scaling the system, but on the other hand, it refers to scaling the developer teams. Both things are easier to achieve by decoupling the system. In a container structure, containers are independent of each other or just decoupled. They are decoupled by load-balancers as well as by predefined APIs (Application Programming Interface). That isolation allows the scaling of containers and allows small teams to work on microservices. The latter comes hand in hand with abstracting the infrastructure. That means no developer is working on a specific machine, as well as the cluster of machines is portable between different cloud environments.

Everything together makes the system more efficient. For example, the container structure enables the possibility to save computing resources. The reason is, different subsystems are packed on the same machine. That allows the creation of images that are as granular as possible to only include the needed functionality. Changing that packing functionality for test systems, potentially reduces the development costs as well [BBH18].

Pod

A *Pod* can be one or more containers that get grouped into an instance. Such a *Pod* is the smallest deployable unit, so the design decision of grouping specific containers into one *Pod* should be considered beforehand. For example, it is questionable to deploy a database and a front-end on the same *Pod*. Depending on the plan to scale, a horizontally duplicated database leads to problematic data inconsistencies. However, it can make sense to put two containers onto the same *Pod*. For example, if both should run on the same machine, share the same network/memory, and are scaled together, it can save work [BBH18].

The *Pods* are placed or packed by Kubernetes on a node that corresponds to a virtual or physical machine. Those manifest the cluster that a specific Kubernetes instance takes care of [3]. The Kubernetes design exists to manage multiple *Pods*. For instance, by replicating a *Pod*, Kubernetes can react to changing load. These additional mechanisms are described below.

ReplicaSet

A *ReplicaSet* is a mechanism that allows managing a group of pods. As the term replica implies, it only works for a set of identical *Pods*. For instance, if four *Pods* are required, the *ReplicaSet* ensures that four pods run. A reconciliation loop (controller) checks the number and arranges measures to achieve this. That means, if one *Pod* fails, a replacement *Pod* is added immediately. If five *Pods* of that instance run, the system kills the fifth instance.

[3] https://kubernetes.io/docs/concepts/workloads/pods/

According to Burns et al. [BBH18], *ReplicaSets* are the foundation of robust applications with automatic failover.

Although, the Kubernetes documentation [4] indicates that using a *ReplicaSet* directly makes sense in cases where the pods follow a no-update or a custom update strategy. If this is not the case, other mechanisms offer update strategy features.

Deployment

A *Deployment* is a mechanism that offers more features regarding updating *Pods* or even *ReplicaSets*. As such, the *Deployment* works on a higher abstraction layer than a ReplicaSet. That explains the possibility of Deployments to manage a *ReplicaSet*. However, it is recommended not to edit these *ReplicaSets* manually [BBH18]. For instance, consider *ReplicaSet* which gets managed by a *Deployment* and is restricted to three replicas. When a user tries to scale the *ReplicaSet*, the*Deployment* has still two in its configuration. Therefore, the *Deployment* overrides the change and kills the additional replica.

Managing the update strategy via the deployment, offers the advantage to reduce the downtime of the service to zero [BBH18]. In addition, Kubernetes handles the updating, so it is not needed to do that manually. The kind of an update is selectable in the *Deployment* via the strategy type. One option is *RollingUpdate*, which means both versions will be online for some time. The user should consider this complexity. If both versions are incompatible, the result is a malfunction of the service. However, as stated by Burns et al. [BBH18], that problem exists for every update strategy. Further, it is possible to adjust the *RollingUpdate*. By changing the max unavailable and the max surge, you can indirectly influence the speed of the update or the available resources. With the max surge, you define how many additional replicas are allowed. For example, ten percent max surge means, during the update, 12 replicas may exist. Max unavailable is the counterpart, when this value gets set to ten percent, then only one from ten replicas may be available. It is also possible to directly influence the speed of the update by changing the *minReadySeconds* parameter or *progressDeadlineSeconds*. The first decides which time period the system waits until a new Pod has to be ready to continue, while the deadline determines the threshold to abort an update if reached.

The other option is *Recreate*, which stands for the simple approach to kill the old version and let the *Deployment* replace them with the updated version. In the meantime, the services will be unavailable, and the user will experience downtime.

Node

A *Node* is a physical or virtual machine that is a computing instance of a Kubernetes cluster. The objects described above are placed/packed on such a *Node*.

[4] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

Service

A *Service* is something similar to that what we already called service outside of the Kubernetes terminology. It covers more than one *Pod* or replicas of them. In short, the cooperation between different *Pods*. This composition establishes service, and the *Service* objects resolve one particular issue, how can different *Pods* communicate or reach each other. They do that by calling a REST (Representational State Transfer) service defined by a tuple of IP address and port with IP address and port attributes, [5]. In addition, *Services*, as *Pods*, get their own DNS (Domain Name System) record [6].

Namespace

It is possible to create a *Namespace*, apart from the default Namespace, which always exists. The namespace concept allows encapsulating of *Deployments*, *Pods*, etc., which is helpful in a multiuser environment. An access control Burns et al. [BBH18] helps to avoid abuse or misuse because it is possible to restrict the access. Outside of its assigned *Namespace*, the object is not accessible, but it also does not interfere with with other entities that exist in other *Namespaces* [7].

Labels and Annotations

While the *Namespace* is specific to the desired use-case, *Labels* and *Annotations* in Kubernetes are not. These key-value pairs are attached to objects as *Pods* or *Deployments* [BBH18]. Used rightly, they allow identifying these objects without using the DNS record or IP addresses. This process is controlled by the user. For example, an (Representational State Transfer) has a required syntax, and the user is free to store a revision number, etc., or decide not to. Same applies for *Labels*. *Labels* and *Annotations* ease for example the allocation of replicas to a node.

Jobs

Kubernetes allows the creation of *Jobs* to specify tasks that will create *Pods*. After these tasks terminate, the *Pods* are killed [BBH18]. Further, there is the possibility to establish a *CronJob* [8], that denotes Jobs to run after a predefined schedule, e.g., for backups.

DaemonSets

DaemonSets help to realize *Pod* instances that should run on all or several nodes of a cluster [BBH18]. An example could be a log collection or a cluster storage [9].

[5] https://kubernetes.io/docs/concepts/services-networking/service/
[6] https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
[7] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
[8] https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/
[9] https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

StatefulSet

A *StatefulSet* is something similar to a *ReplicaSet*. However, as the name says, it is specialized for stateful *Pods*. Therefore, it guarantees that these *Pods* stay ordered and unique, which is crucial when an application is stateful. That means data is stored regardless of which form. As a consequence, it is not possible to interchange or exchange *Pods* or more specific requests. In addition, each replica has a persistent hostname and unique index to ensure this. The index incorporates the sequence in which the replicas got created [BBH18]. With that, the order is maintained.

Ingress

Ingress provides an API entry point which allows accessing the cluster from the outside, for example, via HTTP (Hypertext Transfer Protocol). In addition, it serves as a load-balancer for the incoming requests [BBH18].

Usage of Kubernetes

Users send the mentioned requests to the Kubernetes application. Meanwhile, a user of Kubernetes can use *kubectl* via the command line to establish an SSH (Secure Shell - a cryptographic network protocol) connection to send the commands the API provides. It is also possible to submit YAML files instead. YAML is the format of Kubernetes object configurations and a data-serialization language like XML.

With the terminology explained, we left one part out that is crucial for this thesis, the possibilities to scale in Kubernetes. It is possible to scale *Deployments*, *ReplicaSets*, or *Pods* via API commands or writing scripts that execute those commands relying on metrics. Though, Kubernetes has its scaling mechanisms that do this on its own.

Those are the Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler and Cluster Autoscaler [NYK+20]. Both the vertical scaling and the cluster autoscaler are out of scope for us. Therefore, we present only the HPA in more detail because the HPA is relevant in the course of this work.

Horizontal Pod Autoscaler

In the Cloud Computing section, we described how horizontal scaling works. In short, it in- or decreases replicas. The HPA does it automatically with Pods and the help of a control loop. This loop repeatedly checks that the pods are in bounds of the specified minimum or a maximum number of replicas based on a selected metrics. This number of *min/maxReplicas* and the chosen metrics, including a corresponding desired metric value, are part of the HPA definition regardless of being declarative or imperative. In addition, to which object should be scaled, an HPA can scale *Pods* directly or a *ReplicaSet* respectively *Deployment* that manages *Pods* [BBH18].

The next point of interest is, how does the scaling of the HPA works. Regardless of the selected metrics or target values, it uses the following formula 2.1 from [HPA].

Listing 2.1 This listing is an example of how the Horizontal Pod Autoscaler (HPA) scales up or down. The rounding up is not included. E.g., if the result of desiredReplicas is a floating-point number of 5.1, the HPA rounds it to 6 desired replicas.

```
desiredReplicas = currentReplicas * (currentMetricValue / desiredMetricValue)
```

To explain that in more detail, we use the example from the Kubernetes documentation [HPA]. With the fraction looking like that: 200.0/100.0, the resulting factor is 2.0. That factor doubles the current number of replicas. With a fraction 50.0/100.0 resulting in 0.5, the replicas get halved. This ratio follows the simple intuition that the doubled amount of resources are needed if the used metric is also doubled or halved.

As said above, it is possible to specify more than one metric and if so, the metric with the highest number of desired replicas is chosen to scale [HPA]. However, the metric selection and then specifying the desired value is complex. Firstly, there are multiple options to select a default metric, and secondly, that selection influences the value that can be set. For instance, a so-called resource metric is the CPU utilization, and a possible value is an *averageUtilization*. Firstly, average means that the metric becomes averaged between all existing *Pods*. Secondly, the user specifies the ratio in listing 2.1 which gets multiplied with the *currentReplicas*. Besides, it is also possible to select a direct or so-called *targetValue*. It might be that the metric does not even support setting a ratio.

But that is not all. As the name default metric suggests, Kubernetes supports beyond a limited amount of default (memory and CPU) external, respectively, custom metrics. Custom metrics get added manually. For example, [NYK+20] used the HPA with the Prometheus API for custom metrics. One example of such a metric is the average arrival rate of HTTP requests. Prometheus is, like Kubernetes, also an open-source project and part of the CNCF. The purpose of Prometheus is to provide detailed monitoring [Tur18], and [NYK+20] showed that it helps to optimize the performance of the HPA and that the scaling is slower with the default metrics. In addition, the scraping time is an important parameter. The scraping time is the time window in which the values are collected. It is set to 60 seconds by default, for the default metrics and the Prometheus custom metrics. One can adapt them for both, too. However, Prometheus has features as the rate function. That allows being more proactive because it takes the increase over time into account. As a result, the Prometheus metrics are more effective, benefiting from the long 60 sec scraping time but still reacting quick enough [NYK+20].

That is only one example that shows why it makes sense to use the HPA with custom metrics from Prometheus.

2.3 MoSaIC Project

We introduced the concepts of cloud computing and scaling, that are realized by the technologies of Kubernetes and its HPA. Consequently, in this section, we describe the MoSaIC project. The project follows the mentioned concepts and uses Kubernetes and the HPA for the realization.

MoSaIC stands for **Mo**dellierung, **S**imulation und Design selbst-**a**daptiver **I**oT-Systeme in der **C**loud in German, respectively Modeling, Simulation, and Design of self-adaptive IoT-Systems within the Cloud in English. It is a project from the FZI (Forschungszentrum Informatik) and the University Stuttgart together with Bosch as industrial partner.

The demonstrator implements the use-case of an expert tool that observes vessels as an expert tool. That means a relatively constant amount of experts looks at data that comes from vessels. They do that usually with tool support that helps to react in real-time to certain developments, e.g., face problems that cause the vessel to stop. Then remote maintenance for their engines could be possible with the help of the observed data. That is a realistic scenario, for example, instead of a vessel, such a service exists for trucks. See Mercedes Benz Uptime [10], which offers such a service where Remote Measuring is one aspect to avoid sudden stoppage.

However, as said, the MoSaIC project aimed to develop a demonstrator. As the name indicates, this system was set up to demonstrate a realistic use-case, but it is a mock-up or prototype. That means the implementation is neither used in reality nor usable. To realize this prototype a framework was needed that enables the creation of load. For that purpose ProtoCom [11] is used as performance prototype generator [BDH08]. ProtoCom is part of the Palladio toolchain and does a model-to-code transformation. This transformation is helpful in the early design phase, or for cases when the aim is to demonstrate certain aspects while no real-life system is available. The load or resource demand gets artificially created after calibration on the systems it runs on. The load, for example, a CPU demand, is derived from a calculation, e.g., calculating Fibonacci numbers. The measured values are comparable with the predictions made by Palladio. In work from Lehrig and Zolynski, ProtoCom showed that predictions and measurements correlate [LZ11].

In the case of this project, the performance prototype is crucial because the demand of the experiment should correlate to the demand we model. Let us say the demand is 100 ms, then the measurement of how long the demand runs should be 100 ms, too. In the same way, that works for the model. It helps to focus on other factors and to eliminate at least a confounding factor. Therefore, the performance prototype is embedded in a Kubernetes cloud cluster surrounded by the following architecture.

We indicated above, that it is an expert tool, that means the expert users use a REST interface to obtain data from the DataProvider component. This data comes from vessels, which serve as IoT devices. They use an Ingress to send data to the Device Communication component. The REST interface is realized with Spring MVC and runs on an Undertow server.

The MoSaIC system uses the opensource software Gatling [12] to load test the system. Gatling creates load artificially and records the response time with time stamps in a simulation text file. The software creates from that file graphical HTML reports. Gatling uses the JVM and is written in Scala. It also provides a DSL (domain-specific language) to ease the development of load tests simulation scripts for non-Scala experts.

[10] https://www.uptime-info.mercedes-benz.com/
[11] https://sdqweb.ipd.kit.edu/wiki/ProtoCom
[12] https://gatling.io/

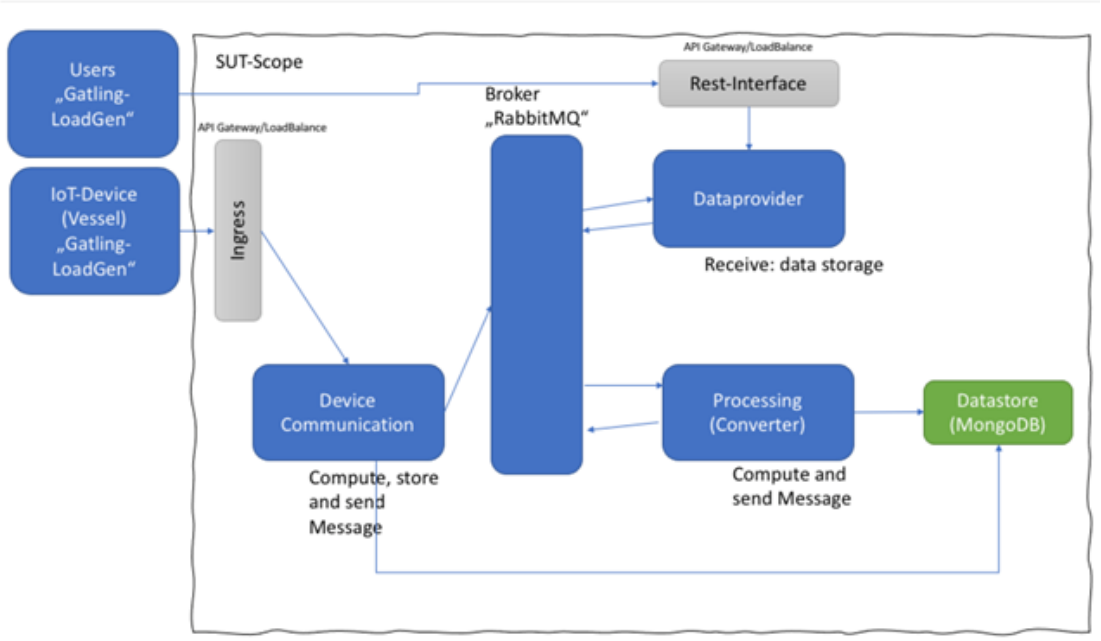

Figure 2.2: Architecture of the MoSaIC system. Illustration from the MoSaIC project.

In between the Processing component comes into action. All three components, which are part of the Kubernetes Namespace demonstrator and have its own Deployment/Pod, are connected to a Broker component. This Broker is a message queue, in this case a RabbitMQ [13], which is the name of the software used to create broker and the message queue. This queue manages the communication between the demonstrator components and allows asynchronous communication between the devices and the database, respectively the end user. The message queues are implemented via Spring Boot, respectively [14]. Spring Boot is a framework written in Java that allows the development of microservices.

Message queues are used except for the possibility that the Device Communication can access the MongoDB Datastore without using the Broker. The Broker and the Datastore are part of the default namespace, as well as Prometheus and the Prometheus adapter.

To make it more understandable what the system does we describe the user, vessel and processing use-case in more detail with the help of sequence diagrams.

We start the use-case for the devices respectively the vessels, shown in Figure 2.3.

[13] https://www.rabbitmq.com/
[14] https://spring.io/projects/spring-amqp

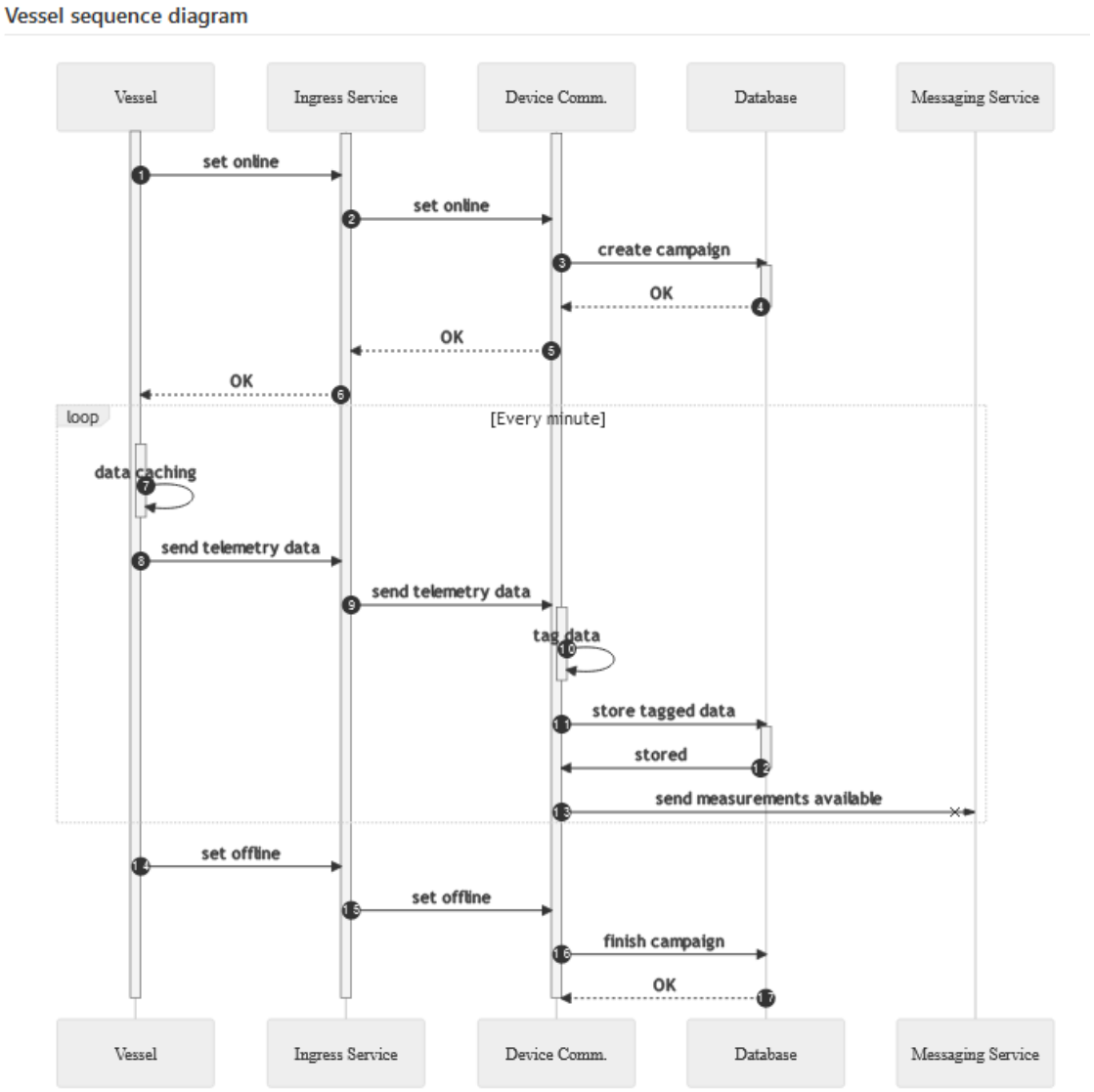

Figure 2.3: Vessel sequence diagram. Illustration from the MoSaIC project.

Firstly a vessel is set online, which means through the Ingress and the Device Communication a campaign is created. After that the vessel measures its data, this is done in a loop and repeats the following process: Cache the measurement data in a one minute time window and send the telemetry data through Ingress to the Device Communication. Here the data is saved in the Database. In addition the status that the measurements are available is pushed via messaging.

Once the vessel is set offline, the corresponding campaign is finished.

If now an expert user wants to retrieve the data of the vessel, the use case works as follows. As depicted in Figure 2.4 the first step of getting the data is to lookup the corresponding ID's of the Measurement campaigns. The idea is not to get the data directly for the Database. In the next step

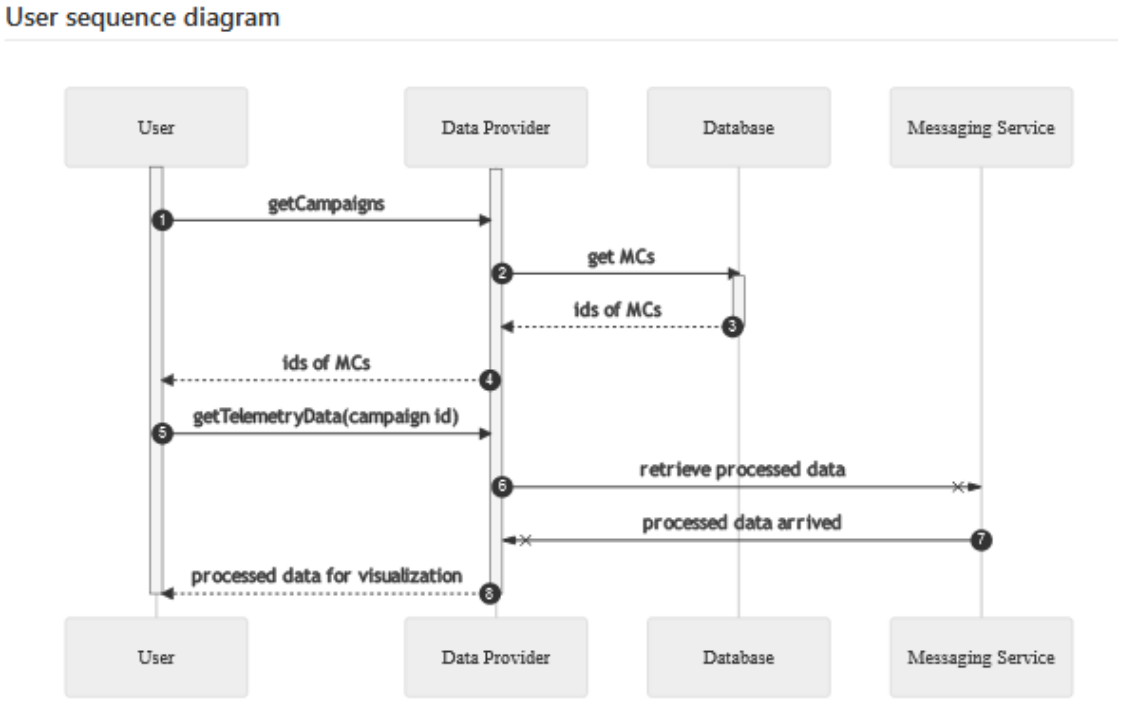

Figure 2.4: User sequence diagram. Illustration from the MoSaIC project.

the ID's are used to get the desired telemetry data via messaging. As this data should be processed, it comes from the Processing component and is then ready for the visualization. What happens in the meantime is shown in the following Figure 2.5.

The data Processing component gets its data from the message queue as shown above in Figure 2.3. As depicted in Figure 2.5 the next step is to check if the ID of the measurement campaign is already in the cache, that happens when the vessel already sent telemetry data. If that is not the case and the vessel sent for example for the first time, the data is retrieved from the Database, is converted and put into the cache. Then the Data Processing sends a message that the processed data has arrived. The latter shows what happens when new data arrives, then the data is converted again and pushed into the cache.

As described above, the MoSaIC project uses the HPA scaling mechanism, which had the target to scale on the development level, the demonstrator Pods.

Regarding the underlying hardware structure, the demonstrator is hosted on the bwCloud [15] and the project has access on seven nodes with four CPU units respectively virtual cores. There is no further knowledge on what are the exact underlying resources or how the bwCloud handles load in terms of prioritization or scheduling. The architecture is as such, that the demonstrator works on VMs and virtualized nodes.

[15] https://www.bw-cloud.org/

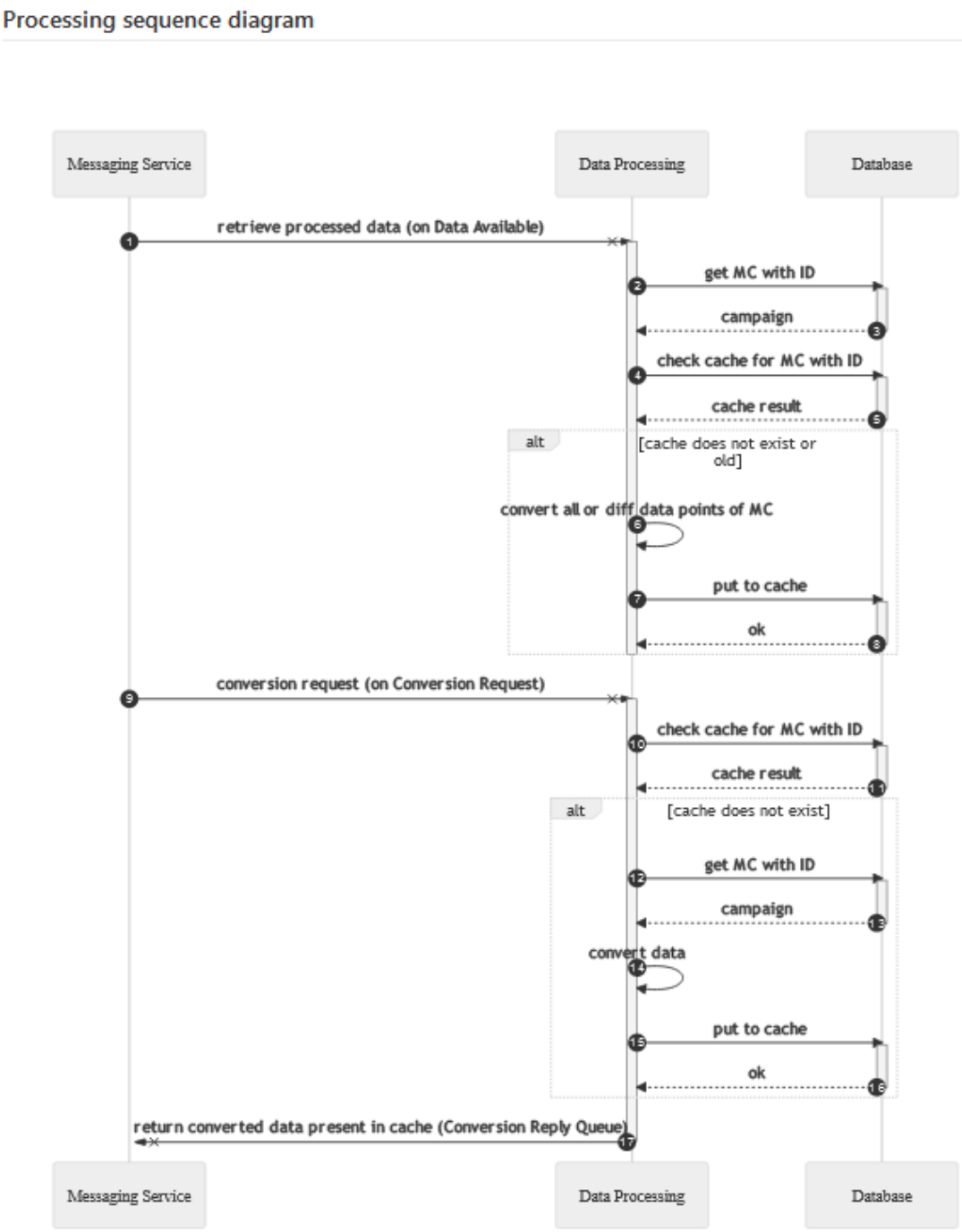

Figure 2.5: Processing sequence diagram. Illustration from the MoSaIC project.

2.4 Architecture-based Simulations

We proceed now with the simulation part of our work. In opposition to discrete event-based cloud simulators as CloudSim [CRB+11], architecture-based respectively model-driven simulators as Palladio [RBH+16] have the advantage that they consider the bigger picture. That is beneficial because when an error occurs after the implementation of a system and that error is traced back to a wrong design decision, changing the architecture at this point has consequences on the existing implementation [RBH+16]. That requires additional and costly work as a consequence. An analogy would be an airport where the building architecturally does not comply with the requirements. For example, an inspection after the construction shows that the hallway is too tight for the expected amount of passengers. Although, the software is hardly comparable with a building because nothing gets demolished physically here. It becomes clear that such changes in architecture are not without consequences here either. For instance, when such a change in the architecture affects many other places in the source code. Therefore that should and can be avoided if the architecture and design decisions get investigated before the actual implementation.

That is not different for cloud systems because that is not isolated and consists of multiple parts/layers composed into a system. Though, the separation of concern can be part of a discrete event-based simulator as well. For example, CloudSim distinguishes between Cloud Provider, Application Provider, and End-User. And of course, CloudSim is specialized in modeling cloud-specific details, as provisioning strategies [CRB+11], but it still lacks to capture the architectural depth.

However, the benefit of architecture-based simulations is the capability to represent the actual architecture of such a complex cloud system and the different stakeholders/roles that act in it. Each system part has its layer rather than only having an abstraction of the stakeholder parties. For example, for Palladio, it is no problem to represent structures as containers or microservices and how they interact. In addition to scaling policies, we described above how many aspects are worth considering in the design phases of a scaling policy. It makes sense to include the architecture of the whole system, especially with the Palladio plugin SimuLizar [BBM13], that added the ability to model self-adapting systems. Event-based simulators, as CouldSim [GSFP20], do not have this ability. Nevertheless, CloudSim [CRB+11] and his extensions [ATTG21] have shown that they can cope with cloud-specific details, as provisioning or autoscaling. For Palladio and its extensions, this is more an open question, which is one reason why we want to investigate that.

2.5 Palladio

To describe the architecture-based simulator Palladio in more detail, we start with the idea behind it. The Palladio approach describes an approach that uses different models (repository, system, resource environment, allocation, and usage) to capture these concerns, and by doing so to foster the understanding of design decisions and their impact [RBH+16]. The aim is to improve the architecture with the help of simulations by allowing to weigh up different design alternatives. To do that, Palladio consists of a component model (PCM), a domain-specific language, and the Palladio bench. The latter is available as an Eclipse plugin to apply the component model and make it usable with the help of graphical editors. Its name is Palladio Bench, and this is a tool that runs

on eclipse to create these models with the help of Sirius graphically. It works out of the box when you download the provided all-in-one product from their homepage [16]. The required Java version is higher than 11. Though, we experienced error messages during the start-up using JAVA 16.

To describe briefly how a system is modeled with the PCM DSL, we take a look at Figure 2.6.

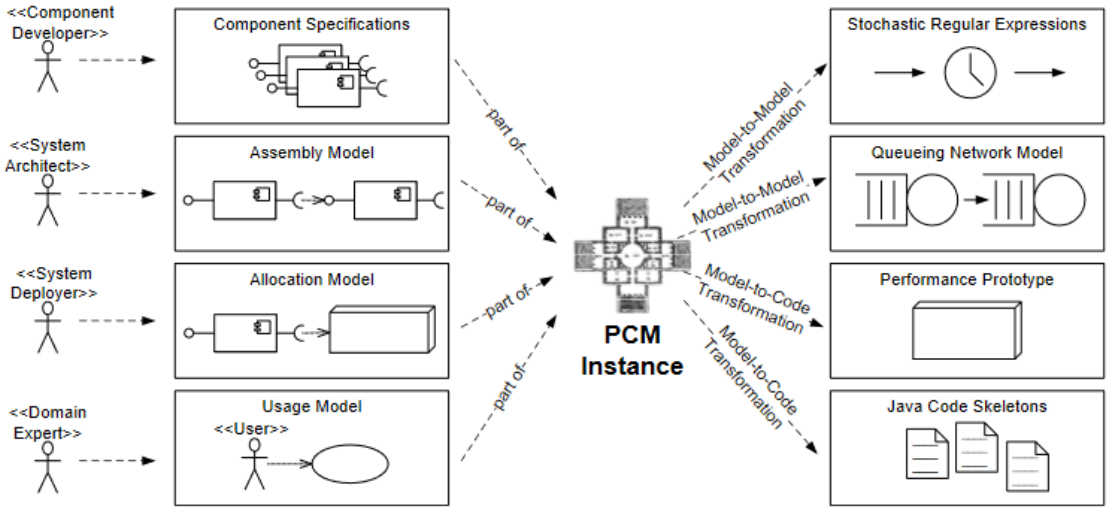

Figure 2.6: Different roles of Palladio, picture from [RBB+11].

We can see in the 2.6 graphic the separation of concern. Each part of a system is separated, and the corresponding stakeholder, respectively developer, can independently model it. We will describe each of the model entities in more detail in the chronological order they get modeled.

2.5.1 Repository

The component developer specifies the individual component in the Repository model. That covers the interfaces of these components, including the methods of these interfaces. The interface specification is the abstract method body, or more precisely, return types or parameters. Return types can be void, which means none exists, and parameters have a name and a type. The component interface contains this method structure or signature, as its name is in PCM. A component then can provide an interface or require one. Require means that the methods of the interface are available to be externally called. If the component provides an interface, then the component can model a SEFF (Service Effect Specifications) for all methods the interface comprises. A SEFF represents the behavior of this method, or in other words, implements the method.

[16] https://www.palladio-simulator.com/tools/download/

Service Effect Specifications

To implement the method, a modeler can specify calls or actions. Actions can be either internal, so they issue a resource demand, or external. An external call means invoking a SEFF from a different component that requires the interface. Additional elements of the SEFF are the control flow actions as a branch, fork, or loop action. To use a fork action means a paralleled behavior becomes synchronized. Within a branch action, it is possible to branch via a condition or probabilistic values. Another concept that is available for components is Passive Resource actions. A Passive Resource has a limited capacity of tokens that can be acquired and released in a SEFF [RBB+11]. The concept is to model a critical section. That means this section is only accessible if a token is available to acquire.

2.5.2 Assembly

With the components defined, the system architect can proceed with Assembly/System model. It represents the interaction between the components and the interfaces. The interfaces that appear in the Repository model are available to the outside. That makes it possible to interact with the outside, depending on whether being a provided or required interface. Such an interface is delegated with the help of a fitting connector to an Assembly context inside the system. In general, an Assembly context represents a component with its interfaces. Those contexts inside the system connect with connectors, which is no delegation because the connections only work from required to provided interfaces.

2.5.3 Resource Environment & Allocation

The system deployer is responsible for the available hardware, how this hardware connects via the network, and what runs on the hardware. Therefore Palladio has two models for the system deployer, the resource environment where he models the first two points and the allocation model where he models the latter point. The resource environment consists of resource containers that have a CPU with scheduling, cores/replicas, and a processing rate. Further, a resource container can have an HDD or a delay. Moreover, with a Linking Resource, the capability of the network connection in terms of latency and throughput is modeled. In this model, the hardware is not necessarily physical. It is also possible to model virtual hardware as a VM. In the allocation, the deployer allocates the components on the available resources. That means placing an assembly context (component) on a resource container.

2.5.4 Usage Model

Last but not least, the domain expert specifies how users use the system in the Usage Model with Usage Scenarios for each interface. A Usage Scenario has exactly one workload that can be closed or open. The difference between both types is that the open workload only has the property inter-arrival rate with an unbound number of users, while a closed one has the properties population and think time. The population is the number of users, and the think time is the time between successfully finishing one request and re-entering for the next. Both think time and inter-arrival rate have in

common that they are random variables [RBB+11]. The inter-arrival rate, the time between different users arrivals, is related to the think time. The Usage Scenario has a behavior where interfaces are called that are available to the outside world. Such that they fit the use case, e.g., using control flow actions as loops or branches. Moreover, more than one Usage Scenario may be modeled for one system, regardless if they share the same workload type or not.
All described models together constitute a PCM instance, but this is not complete. It is possible to model even more. In the description above, we limit ourselves to report that what is relevant for this work.

As shown in Figure 2.6, the user has multiple options to continue with a PCM instance. For instance, he could create a ProtoCom Performance Prototype as described in the MoSaIC section. However, we are interested in the performance predictions, shown as model-to-model transformation. These performance predictions include Quality of Service (QoS) metrics, for example, the performance metric response time. Furthermore, Palladio offers reliability, maintainability, and cost metrics too. An analyzer or simulator executes this simulation. However, this is a bit more complex. Palladio has more than one analyzer, though, for instance, SimuCom or SimuLizar. For this thesis, the SimuLizar simulator and its superset the CloudScale approach [BBL17] project are relevant because they added the capability to model and simulate cloud systems.

The CloudScale approach realized that by extending the PCM with a Usage Evolution, Monitors/Measuring Points, and Self-adaption Rules. These extensions allow modeling a dynamic workload. That is a necessary condition to model and simulate an elastic cloud system. Apart from the dynamic workload itself, the other elements are needed to monitor the state of a system and then self-adapt by increasing the resources.

2.5.5 Usage Evolution

The Usage Evolution links a dynamic workload evolution with the static Usage Model. Above we described how the Usage Model is specified. But it is not possible to model the evolution over time. Therefore it was necessary to add that possibility by using the Descartes Load Intensity Meta-Model (DLIM). The DLIM allows describing a varying load intensity of users respectively their arrival rate over time in a closed workload. Originally, DLIM comes from LIMBO [KHK14] and was integrated in Palladio by [LB14].

2.5.6 Monitor/Measuring Points

A Measuring Point allows specifying what to measure in a simulation, for example, the CPU of a resource environment. In that case, a potential metric is the utilization of that particular CPU. But to specify such a metric for a Measuring Point, a Monitor with a measurement specification is needed. The Monitor points to a Measuring Point and can have more than one measurement specification.

The measurement specification has two properties, a metric and a boolean, defining whether the specification can trigger self-adaptions. Setting this to true allows the creation of scaling policies.

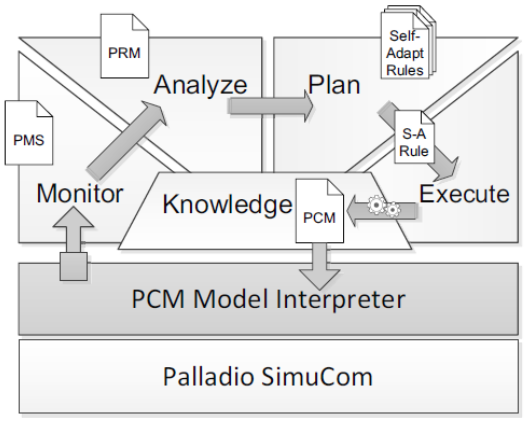

Figure 2.7: MAPE-K feedback loop according to [Mur04], labeled with the SimuLizar architecture, from [BBM13].

2.5.7 Self-adaption rules

The self-adaption rules are the missing part of creating a scaling policy with Palladio. We already described the dynamic workload and the monitoring, but not how the reactions on it are implemented.

As mentioned above, the simulator that works with the extended PCM is SimuLizar. SimuLizar extends the SimuCom and adds the capability to cope with self-adaptive systems in transient phases. A transient phase is a transition, e.g., the interval between two phases, where, for example, the load changed. The self-adaptive system complies with the MAPE-K feedback loop from [Mur04].

The graphic 2.7 illustrates what the acronym MAPE-K means and how it influenced the concept of SimuLizar. To explain that in more detail, there is a Monitor (Palladio Measurement Specification - PMS) that monitors a metric respectively measurement. The measurements are analyzed if they exceed a predefined threshold (Palladio Runtime Measurement - PRM). In the case of exceeding, a self-adaption action is planned, and in the last step, the transformation is executed. In all of these steps, the knowledge base about the system is accessible for the four steps [BBM13].

Options to realize Self-adaption rules are henshin diagrams, story diagrams (sdm), or QVTo scripts [Bec17]. We focus on self-adaption rules that are specified in QVTo because it offers more options to specify self-adaption rules. QVTo is part of the model transformation language QVT. The little o stands for operational [17].

In the end, for the simulation, besides the allocation and the usage model, the user can optionally specify the Monitor Repository file, a Reconfiguration File folder, a Service Level Objectives file, the Usage Evolution file, and Action Model file for the (Transient Effect Analysis).

[17] https://wiki.eclipse.org/QVTo

The latter comes from Stier and Koziolek [SK16], who extended SimuLizar further to include transient effects, which occurs as overhead when an increasing load leads to the provision of an additional VM, as the cost of the adaption. Becker et al. [BBM13] assumed that the self-adaption has none of such additional cost, which means that Stier addressed this limitation with his work.

2.6 Explainability

Software explainability respectively explainable software is a growing research area as the authors book the authors of the book “Self-aware computing systems” demonstrate [Kou17]. They delivered fundamental research, for example, the definition of what a self-aware computing system is. Such a system should be able to capture knowledge about its purpose and reason about that knowledge to decide about future actions to fulfill its purpose. That includes generating knowledge that explains the user or operator its decisions.

Such knowledge is crucial for a cloud system that takes decisions autonomously and relies on a scaling policy. Stakeholders need to understand the system behavior to use or maintain it. To improve the scaling of a cloud system, it is helpful to trace the actions triggered by the policy. [GLV19] summarizes research on that for cyber-physical systems (CPS). These systems do complex computations, use data respectively signal-processing for control tasks, and are at some point distributed. Under that definition, most cloud computing systems are also CPS. The Explainable Self-Learning Self-Adaptive Systems report from [GLV19] proposes retracing adaption decisions to achieve explainability. That includes their trigger, the context, the chosen self-adaption rule, and the expected actual effect.

3 Related work

The related work touches on work that pursued a similar aim but has not sufficiently investigated the prediction of scaling policies. We focus on other scaling simulation systems, like AutoScaleSim. Furthermore, we take a look at work that has been done with Palladio to predict scaling policies in self-adaptive systems.

3.1 AutoScaleSim

The simulation toolkit for auto-scaling of cloud applications AutoScaleSim, from Aslanpour et al. [ATTG21], has the aim to fill the gap that existing simulators lacked in support for auto-scale mechanisms. They used CloudSim as the foundation and extended it so that the autoscaler supports the MAPE-K concept. This functionality does already exists in Palladio. The main entities involved are the cloud provider, the application provider, and the end-user. The MAPE-K concept works between the application provider and the cloud provider. However, the cloud provider is allowed to handle monitoring, analyzing, and executing the system. The performance metrics that were collected through AutoScaleSim are depicted in Figure 3.1

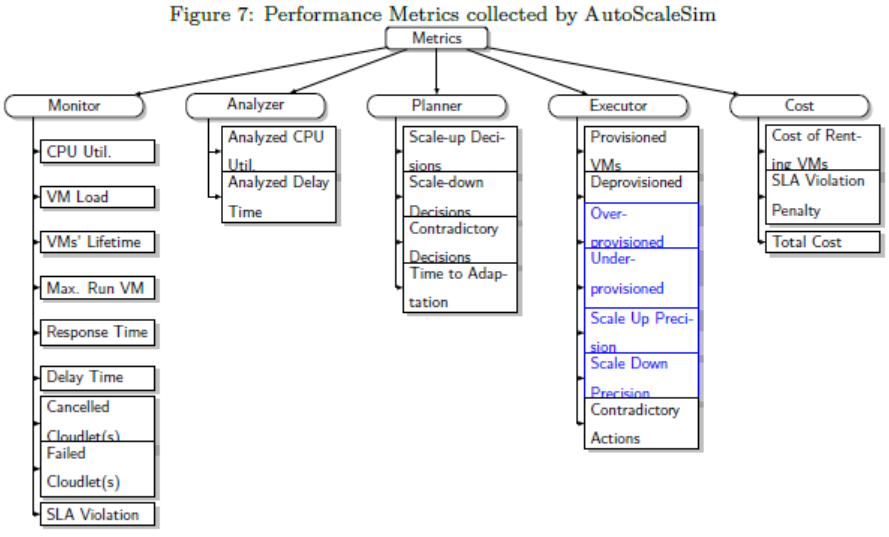

Figure 3.1: AutoScaleSim performance metrics, from [ATTG21].

They deployed and scaled their experiment set-up with OpenStack to validate their work. They used traces from Wikipedia as the workload of users over time. In total, the traces were collected over a time of 211 minutes. They investigated the tail latency, which refers to outliers of very long

latency. That means the analysis looked at how auto scaler influences the tail latency. To do that, they analyzed the 90th and 95th percentile of the response time in their simulator and the testbed. Then they changed the scaling interval to two, four, and eight minutes. Furthermore, they used instant and predictive analysis methods and used a threshold tuning of 70 %, 80 %, or 90 % for the utilization.

As you can see in Figure 3.2, the simulator is capable of predicting trends when changing the interval but misses the tail latency by a lot. In addition, it overestimates the effect of the eight-minute scaling interval on average response time.

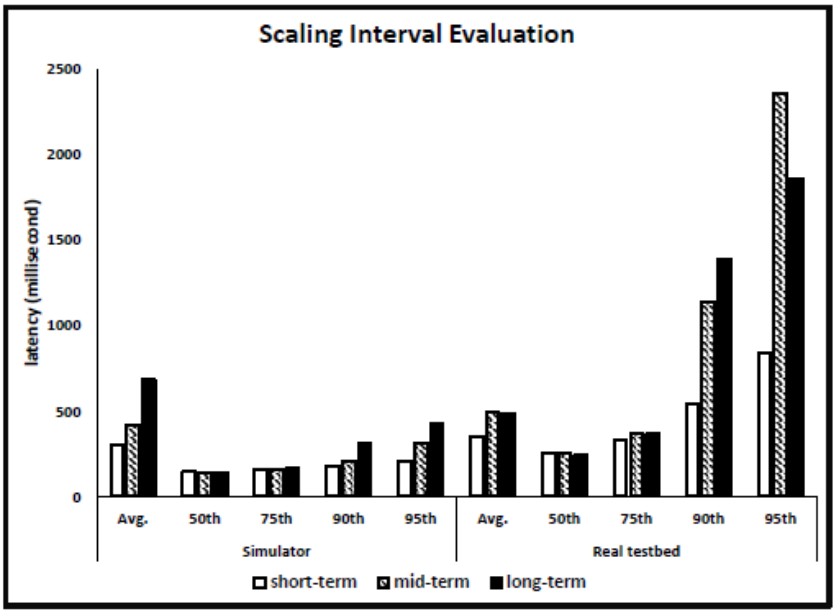

Figure 3.2: AutoScaleSim scaling interval evaluation, from [ATTG21].

As visible in Figure 3.3, the AutoScaleSim underestimates the effect of the predictive analysis. The reason is that a predictive approach can react significantly faster than an instant method. The result for the average response time is still accurate, even if the difference in the simulation is smaller than in reality.

The utilization threshold tuning results in a significant difference regarding the tail latency, see 3.4. However, regarding the average response time or the 50th and 75th percentile, the simulator is accurate. Even if we consider that the trend is not correctly predicted. The tight (90 % utilization) does perform considerably well for the 50th and 75th but is beaten on average.

Even with some limitations, the accuracy of 81 % (mean absolute percentage error) is described as reasonable for such variable environments by Aslanpour et al. [ATTG21]. It is limited to horizontal scaling, not able to model a sophisticated system beyond web applications, as it is a non-architecture based simulator extending CloudSim.

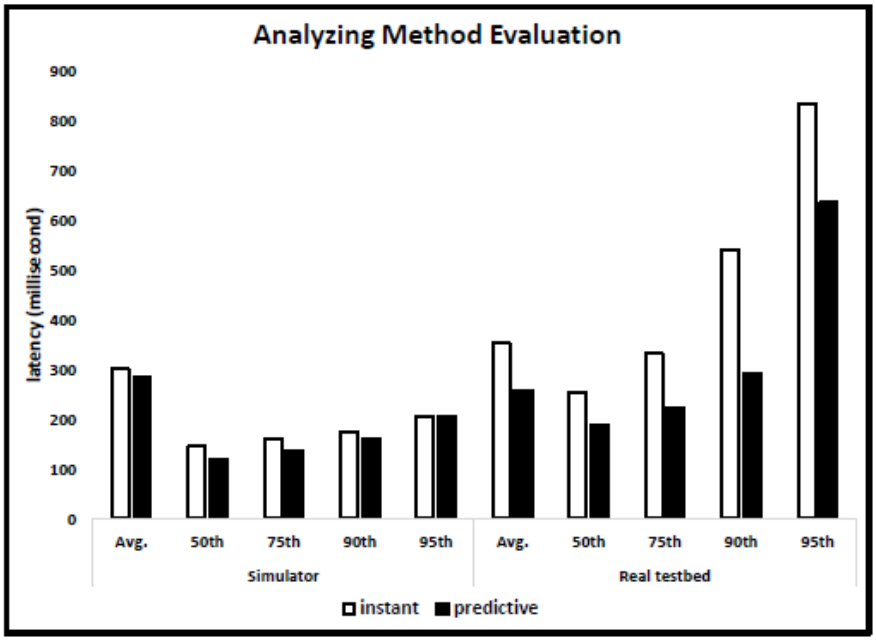

Figure 3.3: AutoScaleSim analysis method evaluation, from [ATTG21].

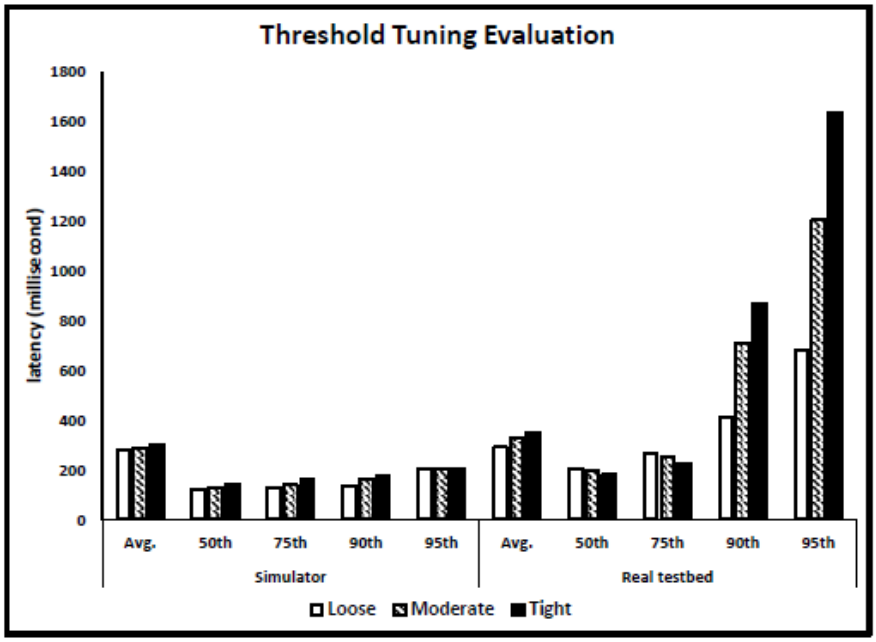

Figure 3.4: AutoScaleSim threshold tuning evaluation, from [ATTG21].

3.2 SimuLizar Analysis

The effectiveness of SimuLizar was investigated multiple times. The analysis from its introduction by Becker et al. [BBM13] showed that self-adaptive systems in transient phases can be modeled. The performance prediction is also accurate enough to differentiate between design alternatives. The accuracy that would allow to do that was set to a value less than 30 %, which SimuLizar kept.

However, at the time, SimuLizar was limited to a static Usage Model, but this limitation was addressed with the CloudScale approach from Lehrig and Becker [LB14], which means that scalability, elasticity, and efficiency analyses are supported now. This was backed by the case study done by Becker [Bec17].

Stier and Koziolek [SK16] did two experiments that demonstrated the accuracy of performance perditions regarding horizontal scaling for the used example system. That was improved by the extensions for the transient effects, e.g., start-up time of scaling action. It also helped to identify additional design flows in the scale-out conditions. In addition, two experiments showed that the approach helped to increase the accuracy for a horizontally scaled application. Figure 3.5 visualizes the reduction of the prediction error. The experiment was executed on an IaaS OpenStack cloud and a reference system that hosted media web-based.

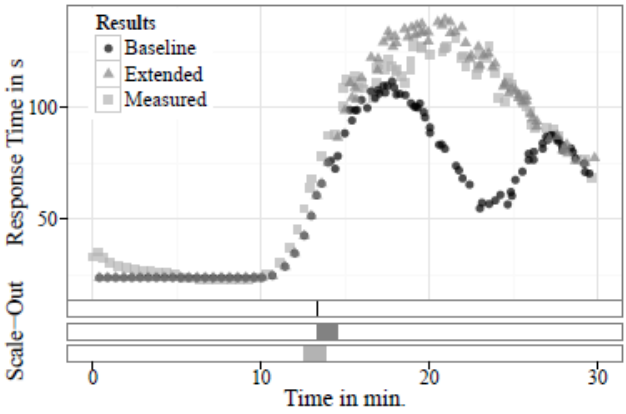

Figure 3.5: Experiment A, from [SK16].

In addition, there was a case study that investigated the Applicability of Palladio for Assessing the Quality of Cloud-based Microservice Architectures [KBB19]. The reference system was a cloud application deployed on AWS. However, they did not investigate scaling policies because the company where the case study was conducted had no scaling policies in place. Therefore they could not investigate its accuracy. Further, they even deemed it infeasible to model elastic scenarios with message queues because they found no way to react to their status. The work showed that Palladio can model a complex microservice application and get a reasonable accuracy. Although, as Figure 3.6 shows, the accuracy became worse with an increasing number of users.

Another related work that investigated something similar to us was done by Klinaku et al. [KHB21], which evaluated Architecture-based Scaling Policies for Cloud Applications.

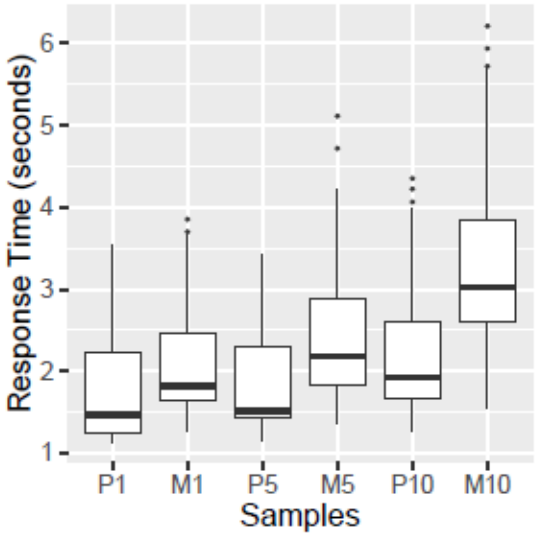

Figure 3.6: Evaluation from [KBB19].

Before they implemented scaling policies, they did a workshop to determine scaling policies along the dimensions we presented in our foundations. The dimensions were validated during a workshop with three experts from academia and the industry. Further, they created their scaling policies with the help of a self-defined Scaling Policy Model that included *ScalingTrigger*, *TargetGroup* (what to scale), *AdjustmentType* (how to scale), and *PolicyConstraints*. In the end, they came up with three policies:

- CRP based on Overall Response Time Architecture Aware
- CRP based on Utilization Architecture Agnostic
- DRP based on Queue Length Architecture Agnostic

The first letter of the abbreviation shows that two from three policies are centralized (*C*RP). The other is decentralized (*D*RP). All of them were reactive. They reached an agreement to use scraping time and quiescence period (60 secs) as a constraint to avoid oscillation. Furthermore, the policies were bound to a specific number of resource containers. The acronym follows the matrix from Figure 2.1, at least for control and anticipation. Their performance can be viewed in 3.7, while they all perform better than no scaling policy (np), there is only a slight difference between them. The difference comes from the fact that the crpu policy reacts slightly faster than the others. The reason is that utilization of 70 % is looser than the other thresholds. Although, the difference between them was considered as not statistically significant by [KHB21]. The model where the policies were applied came from [KBB19], resulting in the fact that no measurement data is available to assess the accuracy of the modeling.

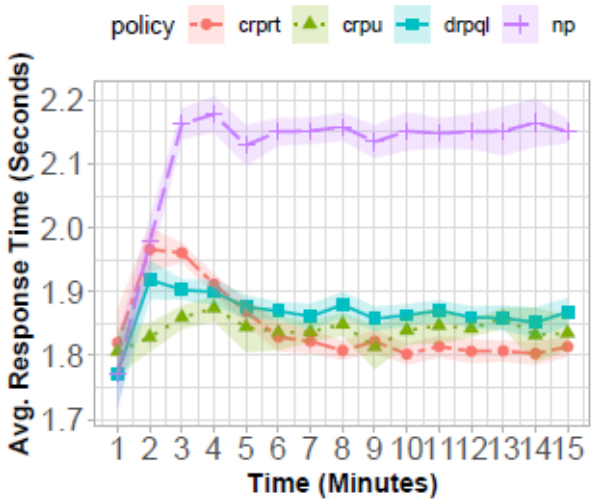

Figure 3.7: Mean response time of the policies, as inspected by Klinaku et al. [KHB21].

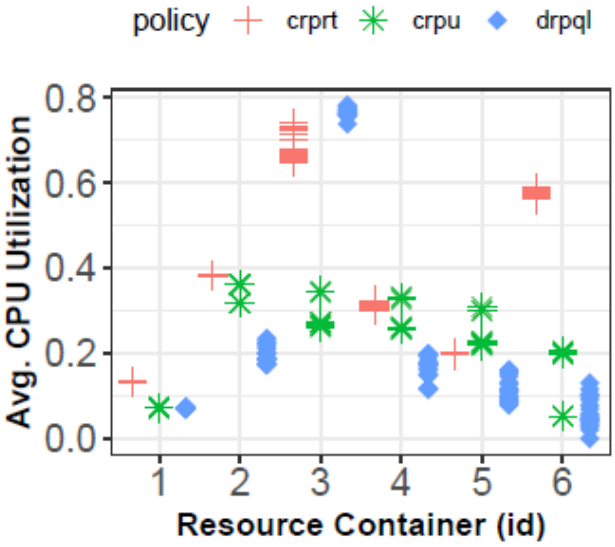

Figure 3.8: Mean utilization during the experiment, as inspected by Klinaku et al. [KHB21]..

The scaling policies on the model site were realized using QVTo, which is problematic because of the effort. Another problem is that no meta model to specify a scaling policy existed. The latter is problematic because the unknown execution semantic can cause arbitrary effects. Therefore a meta model for defining PCM-based Scaling Policies for Cloud Applications was deemed as future work, as well as an accuracy analysis.

4 Research Design

This thesis aims to investigate the feasibility and aiding the explainability of scaling policies using architectural-based simulations.

That means to model scaling policies for an architectural-based simulator - Palladio - and run the simulations. With the simulation results, we want to answer if it is feasible to model scaling policies and if the results aid the explainability. Both questions touch on the research gap - is Palladio ready to simulate scaling policies.

4.1 Research Gap

Although related work touched a similar topic, neither Klinaku et al. [KBB19], nor Klinaku et al. [KHB21] assessed the accuracy of the self-adaption/scaling policy modeling mechanisms of Palladio. As described in the related work, Klinaku et al. [KBB19] showed that it is feasible to create a cloud microservice with PCM and run the simulation with a satisfying number of accuracy. But they did not investigate the accuracy of scaling policies. Klinaku et al. [KHB21] used the system from Klinaku et al. [KBB19] to show that it is possible to model scaling policies. Though, they could not show how accurate the simulations are because of missing data. Further, Stier and Koziolek [SK16] did an accuracy analysis for a scaling mechanism but not by using a microservice or Kubernetes messaging system. In addition, their scaling mechanism did not fit the scaling policy description introduced by Klinaku et al. [KHB21].

That means the research gaps encompass the accuracy analysis of scaling policies that run in a Kubernetes messaging system and the question if the PCM model respectively simulation aids in terms of the explainability of scaling policies. For the latter, we found no related work which investigated that before.

4.1.1 Research Questions

The research gap leads to the following research questions:

- RQ1 What is the feasibility of evaluating cloud scaling policies through the architecture-based Palladio?
    - RQ1.1 How accurate are the results of the simulation compared to reality?
    - RQ1.2 Is the best-performing configuration predicted correctly?
    - RQ1.3 Is the simulation predicting the best performing scaling policy style correctly?
- RQ2 Can architecture-based simulations aid in improving the overall explainability?

As we described, the missing part regarding the feasibility to model and simulate scaling policies with PCM is an accuracy analysis that demonstrates an acceptable accuracy. Therefore the focus of this thesis is the accuracy of the simulations, see RQ1.1. However, accuracy captured in percent is only one possible perspective and a validity threat for this thesis. We are aware of the model calibration complexity. The virtualized Kubernetes system induces this complexity. To manage this risk, we added a second perspective, see RQ1.2. If the best-performing design alternative is predicted correctly, then the performance prediction is not useless. In other words, if the trend is correctly captured, the results are more robust to calibration issues.

Regarding the second research question (RQ2), we rely on Greenyer et al. [GLV19] and their proposal that retracing adaption decisions aids the explainability of adaption decisions. Therefore we want to investigate if Palladio can provide the retrace ability of adaption decisions after simulating. Of course, this is vague and only an initial step for research. We could not ask domain experts with a survey to validate our findings through time constraints.

Before we conducted our research, we formulated the following hypotheses:

- H1 It is feasible but with some limitations.
- H1.1 We expect better accuracy than the 30 % named by Becker et al. [BBM13] to be acceptable by the performance engineering community.
- H1.2 We expect that the best-performing configuration is predicted correctly. Even if Aslanpour et al. [ATTG21] showed that this is not straight forward.
- H1.3 We expect correct results of predicting the best-performing style, as well.
- H2 We would expect that the model helps to foster the explainability by providing information to trace the scaling decisions.

4.2 Process

The process to answer the research questions has the following parts.

Firstly, we want to select the scaling policy to implement. We do that together with the MoSaIC project. The second part is adding this scaling policy to the MoSaIC system and the model. Our focus is on the model, where we can use an existing model and existing QVTo scripts from other projects. Then we prepare the experiment by selecting a scenario and a load to stress the system through a load-test. The next part is executing the load-test and collecting data. In parallel, we make the model ready and calibrate it. After that, we run the simulation and collect data. Both data sets are the foundation for the evaluation of the research questions see Figure 4.1.

The load test of the experiment is a special case, therefore we will explain this item in more detailed. It has the following sub-items:

- Scalability
- Elasticity

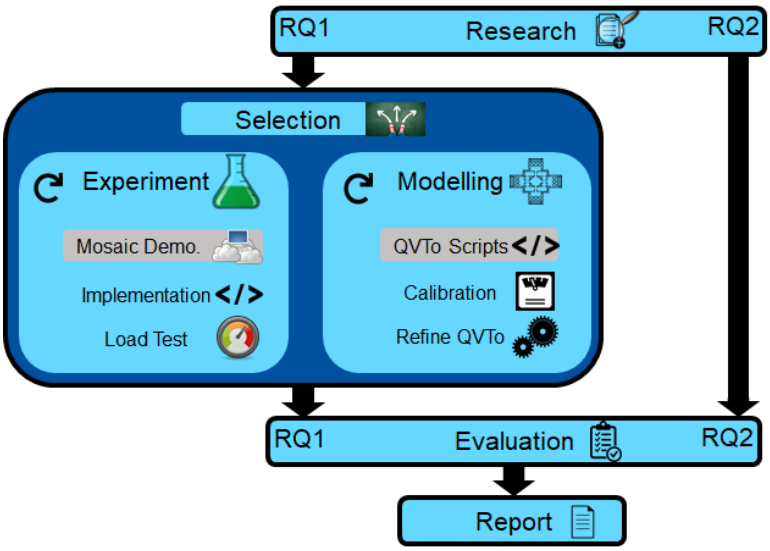

Figure 4.1: The process of this thesis.

The need for an SLO and to benchmark the computational capability of the system requires the execution of a scalability experiment before doing an elasticity experiment. We need it because it was unclear which load results in a 100 % CPU utilization. In addition, we had only a rough SLO available from the stakeholder. This SLO was partly usable because it was generic and not suited to our specific setting. A fitting SLO is important for choosing an adequate number of vessels for the elasticity experiment. Otherwise, we would risk running a setting that does not stress the system. That is problematic because scaling is only reasonable when one node cannot handle the load anymore. The scaling action is an action to keep the SLO.

From the available use cases of the MoSaIC system, we selected the vessel scenario shown in Figure 2.3. The plan is to run a ramped-up load scenario with 100 vessels as baseload. Over 15 minutes, we increase the number of vessels in four steps. The increment per step would be 100 additional vessels. That results in 500 vessels are sending data at the end of the four ramp-ups. And repeat that for four available nodes but with a higher amount of vessels.

This so-called Strawman (linear search - increasing with a constant factor) approach is deemed as inefficient by [SMC+08], who also describe better alternatives as the Bbinsearch algorithm. Binsearch is an adaption of the binary search, that doubling the rate. The intention behind a scalability experiment is to save time in the experiment process. That does not only apply to the elasticity experiment. It also applies to the scalability experiment. It is reasonable to use a more efficient method to find the utilization sweet spot or peak rate. The alternative of trying via brute force or the Strawman to find a suitable configuration would cost too much time. Therefore we adapt our plan and opt for something similar. Instead of a linear increment, we double the number of vessels, e.g., starting with 500 and doubling the baseload to 1000, triple it to 1500, pp. That approach ensures to reach the point of 100 % utilization faster. We also differentiate between one and four nodes. As a result, we start with a fourfold higher number of vessels for four nodes.

The target is to observe the response time and the utilization of the system to find out the fitting configuration. This configuration should stress the system close to its maximum. We do not want to over-stress it because that would confound the result. With that knowledge, we can continue with

the elasticity load. The elasticity load should test the auto scaler with two scenarios - a low and a medium. Low and medium refers to the number of vessels. The low case should not stress the system much because we want to observe what happens if the system saturates. The medium case should have an amount that goes beyond the computational capability of one or two nodes. That means the system should do at least one scaling action. However, as the name medium indicates, the system should be capable to process the load without all four nodes.

The motivation for this plan is the use-case, where a new customer joins the services and adds his vessels to the system. Other than in a unprecedented scenario, the provider can plan and control the way the vessels join and how the system configuration should react on the additional load. That is the reason we do not consider a high or extreme scenario regarding the number of vessels. With the result of the scalability experiment, the provider knows the capability of his system and can use that as reference to plan. In such a scenario an accurate simulator is helpful to know upfront which scaling threshold and ramp-up time is to favor in terms of performance.

4.3 Methodology

We assess the accuracy of the simulation by using the metric response time of the usage scenario and the utilization of the measured resources. The response time is the main metric to calculate accuracy by dividing the measurement by the simulation. We use the mean absolute percentage error (MAPE) [KK16], with the following formula:

$$\text{MAPE} = \frac{100}{N} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right| \tag{4.1}$$

Where N denotes the number of data points, A_t is actual, and F_t is the forecast value. MAPE is a suited measure of forecast accuracy and was also used by [ATTG21] to specify the accuracy of AutoScaleSim.

Similar to Aslanpour et al. [ATTG21] we do not solely focus on the mean value of the response time. In addition to the mean, we analyze the 95th and the 99th percentile to determine the accuracy. Other than Aslanpour et al. [ATTG21], we decided to take an even closer look at the tail latency by replacing the 90th with the 99th percentile. That allows judging on the tail latency and the effect of outliers. That is relevant because the mean value can hide the impact of scaling policies. The reason is the transient phases, where the scaling actions are active. In that time window, including the period of the high utilization, the response time is higher than during a steady state. The result is that only a few requests need a long time. Therefore, to assess the quality of a scaling policy the tail latencies reveal the effectiveness of the scaling decision. For instance, if the tail latency of scaling policy A is significantly higher than the tail latency scaling policy B - policy A reacted too late. At least on a high level where the policy shares the same style and differs in the configuration. Hence, we decided to concentrate on adapting one parameter at a time. That means we do not change the cool-down period and the threshold together. On a low level, where the styles of the policies are different, a broader approach is handy. That does not mean ignoring the tail latency. It means checking the efficiency through the nodes used, too.

RQ 1.1 and RQ 1.2 ask for the trend conformance regarding the best-performing policy predicted correctly. However, for that, we should consider the standard variation. The reason for this is we fear that the calibration is problematic for the tail latency. Including extreme values at this point undermines our objective to add another perspective to manage that risk.

The second research question (RQ2) has no specific metric. As described above, we want to investigate Palladio's self-adaption tracing capability. We define that the result allows us to understand why an additional resource is added or removed. That means retracing the values of the scaling trigger values in some graphical form.

5 Results

The results chapter is separated into four parts, how and what scaling policy we choose, the conduction and results of our experiment, the Palladio model results, and the conduction and results of the simulation.

5.1 Scaling Policy Results

The selection of the scaling policy should be done in agreement with the project partners of the MoSaIC project. Initially, the plan was to propose ideas to them during a workshop, as Klinaku et al. [KHB21] did in their work. However, the process was delayed after creating a proposal. Instead of presenting our ideas, we passively followed the discussion from the project partners. This discussion is described in section Section 5.1.1.

5.1.1 Workshop

The workshop consisted of two meetings, separated by one week. During the preparation of the workshop, it became clear that the elicitation is not trivial. The stakeholders agreed that it is difficult to select scaling policies that fit their demonstration purpose. From the model perspective, one requirement was that the policies should be easy to implement and work out of the box. The aim is to model them, which means it is necessary to use metrics/mechanisms that can be modeled with Palladio. Further, it was agreed to start with horizontal scaling to keep it simple. In addition, the industry partner favors horizontal scaling.

Then concerns about modeling the HPA were part of the discussion. The HPA formula is more complicated than a simple threshold-based scaler, but the concerns were related to the Pod concept. In Kubernetes, the HPA scales the Deployments or Pods, which translates to a container. It is possible to consider the Palladio resource container as a Node consisting of several Deployments/Pods. The problematic part here is to model the Kubernetes utilization metric because Palladio only provides it for the Resource container as a whole. With multiple Deployments on one Node and an HPA that scales single deployments, that does not work and leads to constraints on how to realize the HPA. For instance, find a workaround, e.g., isolate the Deployments, or use a different metric like the response time.

Therefore, all parties agreed to firstly select naive policies that scale based on simple metrics like CPU utilization and secondly to use an abstraction layer that that can be modeled with Palladio. The result is a policy that scales the nodes rather than the Deployments or Pods. The result was refined with a wiki page. The collected information was the basis of the second meeting.

At the second meeting, two possibilities were presented, homogeneous Deployment and non-homogeneous Deployment. Homogeneous means that on a single Node all three MoSaIC Deployments run. Here scaling means scaling all Deployments together, as shown in Figure 5.1.

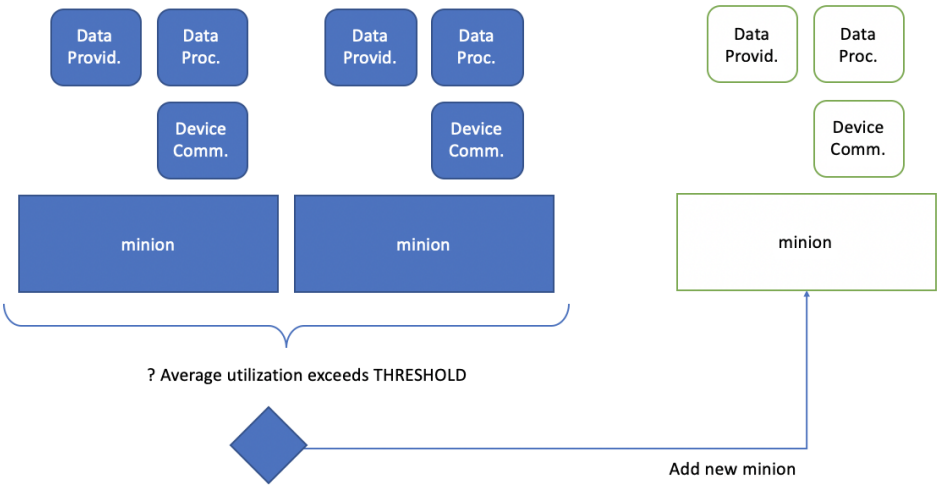

Figure 5.1: Homogeneous Deployment. Illustration from the MoSaIC project.

In contrast, non-homogeneous refers to the fact that the deployments were split so that one node consists of two Deployments and the second has only one Deployment, see Figure 5.2.

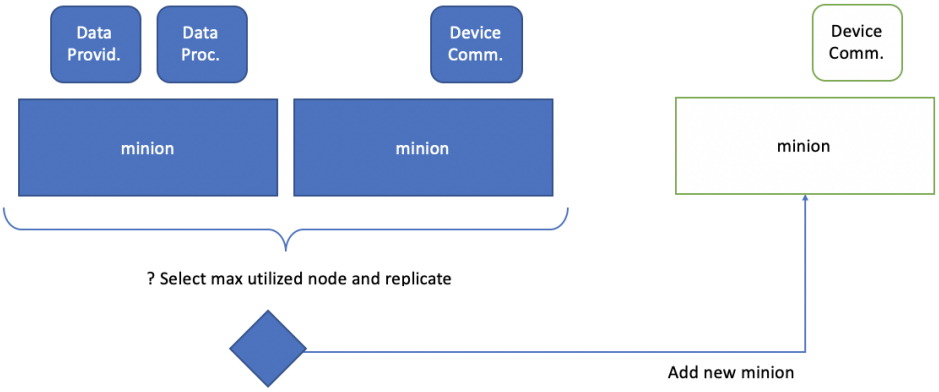

Figure 5.2: Inhomogeneous Deployment. Illustration from the MoSaIC project.

The homogeneous approach is agnostic, in contrast the non-homogeneous is aware. It is aware, when the reason for the arrangement is the architecture of the system. For example, as the Device Communication is considered a bottleneck, it is put on a separate node. Then both nodes are monitored, and the node with the higher utilization is scaled. Hence this is a more sophisticated solution.

Further, it was agreed to implement the scaler by a project partner. That means they decided against using an existing scaler with similar functionality that already existed on GitHub. Apart from that, the implementation details have been omitted. In the terminology of [KHB21], the TargetGroup and AdjustmentType have been set. However, the Scaling Trigger and PolicyConstraints were subject to being determined in the implementation.

5.1.2 Implementation

The scaler that implements the findings of the workshop has the name Node Utilization Scaler. The controller of the scaler is written with the help of fabric8 [1], a Java opensource framework for Kubernetes APIs and later containerized with JIB [2].

The first policy is called node utilization scaler with the following properties. The scaler has an upper and lower threshold that are parametrized. So it is easy to change the thresholds, without having to deploy a new container. For instance, it will scale down if the lower threshold is set to 0.3 and 30 % utilization is reached. If the upper threshold is set to 0.8 and a utilization of 80 % is reached it will scaled up. It monitors the utilization over a window of 15 seconds, where the average CPU utilization is collected and used to reconcile.

The scaler has a minimum number of one node or replica and a maximum number of four replicas. The scaling adjustment is one. That means it scales only one node at a time. Further, the cooldown period of two minutes prevents any scaling action in the next two minutes. To classify the scaling mechanism according to [KHB21] we have a centralized, agnostic, and reactive scaler.

Due to time constraints no autoscaler with an inhomogeneous deployment was realized.

[1] https://fabric8.io/

[2] https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin

5.2 Experiment Results

In this section, we describe the results of the experiment. We split the results into two parts, the Elasticity Experiment preparation section, and the elasticity experiment section.

The preparation for the elasticity experiment is the scalability experiment. We needed the first to find the peak rate of the MoSaIC system. Without the scalability experiment, it would have been impossible to run a specific load that tests the scaling policy and the autoscaler. Therefore we first conducted the scalability experiment with no autoscaler and scaling policy active. After that, we can pick a reasonable load and execute the elasticity experiment. This experiment is described in the section 5.2.3 Elasticity Experiment Conduction.

5.2.1 Elasticity Experiment Preparation

As described in the foundation chapter, we use Gatling to execute the load tests. As already described the use case that will create the load it the vessel scenario from Figure 2.3. It was possible to include also the expert user with his use case shown in Figure 2.4. We decided against that and used only the vessel scenario to reduce the co-founding factors.

We had two possibilities to run the load test experiments, either triggering it locally with a batch script and port forwarding, or directly on the cluster as Kubernetes job. The local solution was instable and the port forward often aborted abruptly. That forced us to use the job solution. That has also downsides but was more stable, when it runs and it was easier to change the parameters of the scenario.

Gatling communicates with Ingress and sets a device online. After the device is online it sends data forever. Between each sending operation of the same vessel, there is a 60-second pause. The load test terminates when the maximum simulation time is reached.

Scalability Experiment Load

The load for the scalability experiment looks like the following. We started with a baseload of y, where y denotes the number of vessels sending data at the start of the experiment. The number of vessels is then increased over four increment steps, by multiply y with the step factor. The step factor for the baseload is one, for the first increment step two and so forth. By a different counting method, the baseload is step one and the last is step five. So that we finish with the step factor five and increase the baseload fivefold. That is the binary search The experiment duration is 15 minutes, with three minutes pause between each increment step. The devices are ramped-up over a period of 5 seconds, that means Gatling will set all the devices online in that time window.

For a graphical representation, compare Figure 5.3.

We start the scalability experiment with y set to 100, 300, 400 and 500 vessels for one node. For that we allowed only one replica in the Kubernetes deployment. To save time we skipped two and three replicas by scaling to four replicas of each demonstrator services (Device Communication, Data Processing and Data provider). For four nodes we increased y to 500, 1000, 1500 and 2000 vessels. That range represents a multiplication by the factor four of the one node baseload.

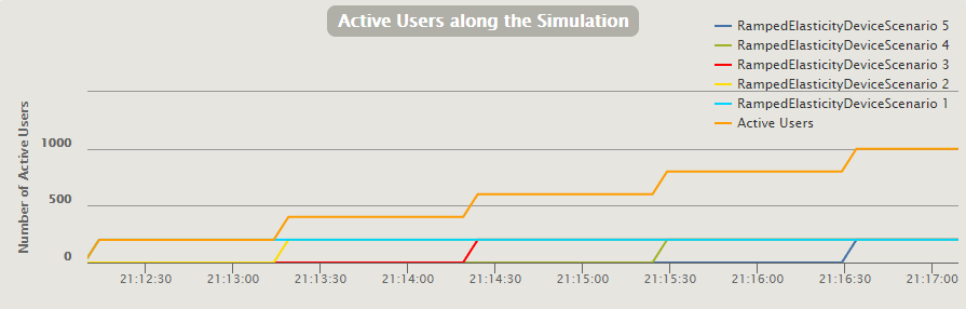

Figure 5.3: Gatling Load for the scalability experiment.

We track the response time but exclude the five seconds ramp-up period. In addition we check the CPU utilization metric of Prometheus after the experiment to validate the scalability experiment.

Scalability Experiment Results

The results of the scalability experiment are the following. The scenarios with the baseload of 500 and 2000 vessels reach the peak rate of the system. The reason why we come to this conclusion is the fact that one node processes 2000 without major anomalies in the response time. However, when one node has to deal with 2500 vessels, the response time increases significantly. Regardless, if we look at the mean or the percentiles, compare with 5.4.

It is close to one second for the percentiles, and the mean response times increased too. The same happens with four times the number of vessels on four nodes. Although the percentiles increase less continuously here, and the response time decreases for 10000 vessels, compare Figure 5.5.

For both cases, the number of vessels does not create an overload situation. Although, the last increment step puts them under load that is reflected in the response time.

After we execute the scalability experiments, we can look at the CPU utilization of the nodes. We can see that response time observations are backed by the utilization numbers. The peak load of 2500 vessels for one node and 10000 for four reach the 100 % utilization time. We consider that 100 % utilization of the system means 400 % of the CPU, because the nodes have four virtual cores. Although, the peaks for some nodes are not completely equal. Figure 5.6 of the Prometheus output shows that the lines do not fully overlap. That means the Kubernetes default Ingress load-balancer seems not to distribute the load equally. A limitation of the Prometheus utilization is the scale of the graphic. It is hard to tell how long the peak load persisted. In addition, we did not observe the influence of the ramp-up and the setOnline invocation.

Therefore we do not think that the peaks show a persisting overload situation, which is also an observation of the scalability experiment. To discuss that in more detail, we refer to Figure 5.7 that shows how the system is stressed. The colorful peaks show the response time percentiles of the sendData method. They do not occur permanently, only for a limited amount of time.

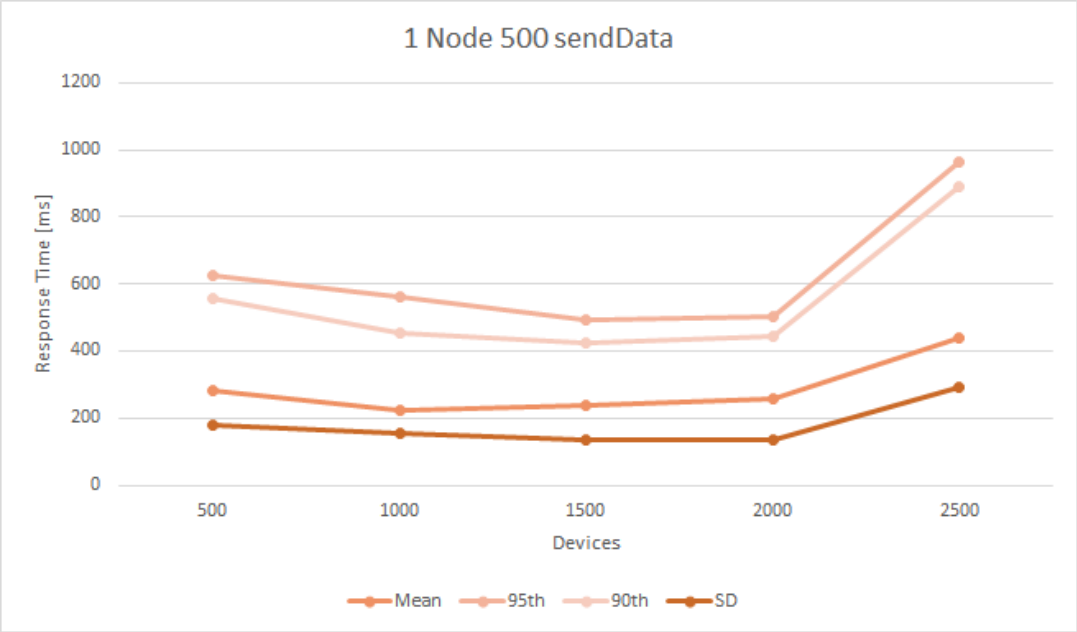

Figure 5.4: Scalability experiment with a baseload of 500 ran on one node.

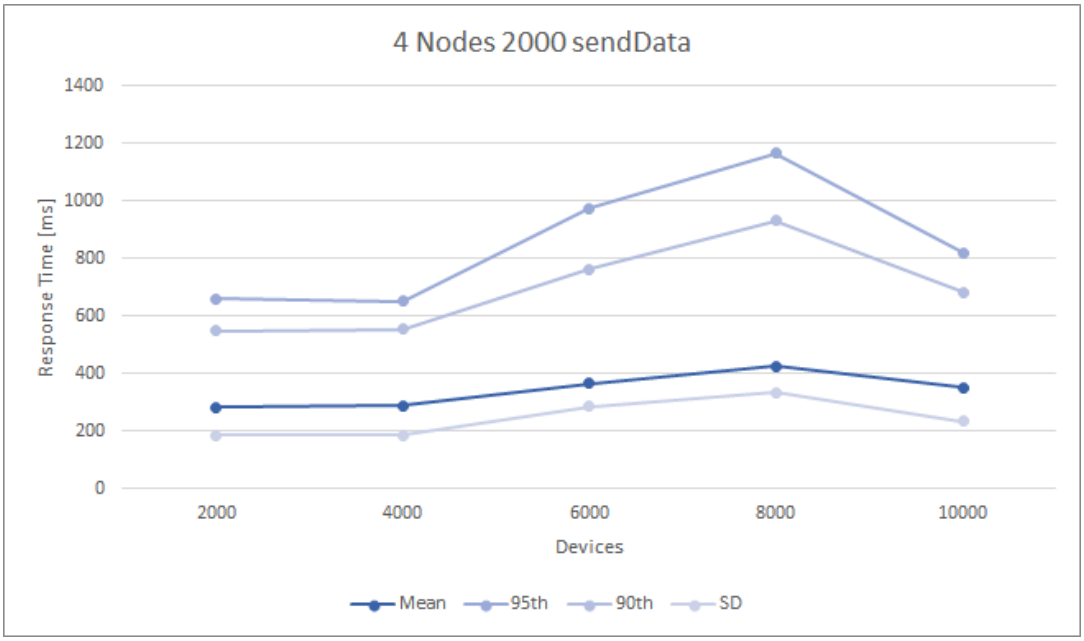

Figure 5.5: Scalability experiment with a baseload of 2000 ran on four nodes.

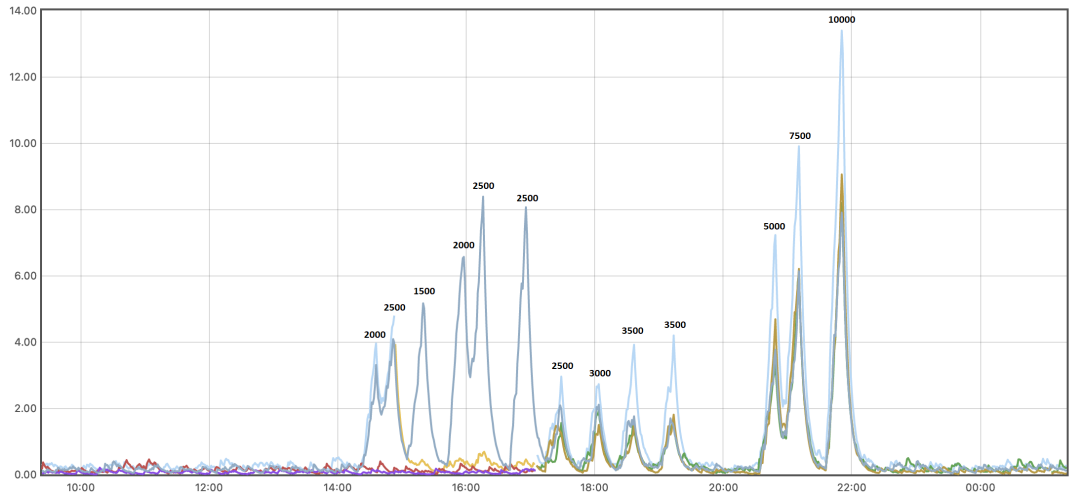

Figure 5.6: Utilization measured by Prometheus, the peaks are labeled with the maximum number of vessels.

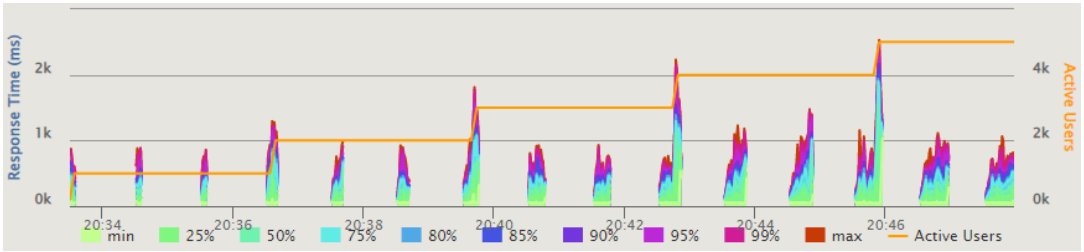

Figure 5.7: Response time percentiles for sendData.

We think the peak utilization comes from the ramp-up of the last increment step. After the last data is sent and the first batch of this step is finished, the system reaches an idle state soon. That happens because all active vessels wait 60 seconds to send again.

For a scalability experiment, this phenomenon is not problematic. Our aim was not to create a constant load. Although, there is a certain uncertainty about what we benchmarked. After seeing Figure 5.7, one would argue we observed that a ramp-up period of five seconds and more than 500 vessels create an overload. That argument is reasonable, but as we excluded the ramp-up time and the sendData executions in that period, the response time analysis from above is not disturbed by that.

We can conclude that we have learned how many vessels would create a sensible elasticity experiment. In addition, the unequal distribution of the send data executions creates noise and is not suited for an elasticity experiment. And that a SLOs of one second response time for the 95 percentile would make sense because that is the ceiling we nearly hit with our experiment.

The exact numbers of the graphical representation can be viewed in the tables A.1 and A.2

5.2.2 Elasticity Experiment Conduction

With the obtained knowledge of the scalability experiment, we can proceed with conduction. That means first determine the load scenario, describe the set-up and show the results.

Elasticity Experiment Load

From our perspective, we had the following requirements for the load: triggering a scaling action to more than one additional load. Then concerning the constraint 300 seconds is theoretically enough to scale three times from one node to four, depending on the utilization from the issued load. However, we decided to run the experiment for 1200 seconds because we wanted to observe a longer time window with an equally distributed load. Therefore we changed the ramp-up time from five to 60 seconds. The reason is after a device is online, it sends data and then executes the next send operation 60 seconds later. With that setting, we guarantee a constant load instead of squeezing everything into five seconds and creating an overload.

That makes sense for two reasons. Firstly, it is realistic. The system use case is that a new customer adds his vessels to the system. Under normal circumstances, a predictable event is planned. As such, we would assume the ramp-up is configured in the same fashion. Secondly, idle phases are bad for our scaling policy or autoscaler. If the implementation would be more robust then it is less problematic. For instance, considering values from a longer period than 15 seconds, then it is likely different. But in this case, the system will oscillate between up and downscaling within the time constraint. Chance decides whether the system is currently under load or idle. For our scaling threshold tuning experiment, this behavior makes not much sense.

The load itself is increased by one increment step, as Figure 5.8 shows.

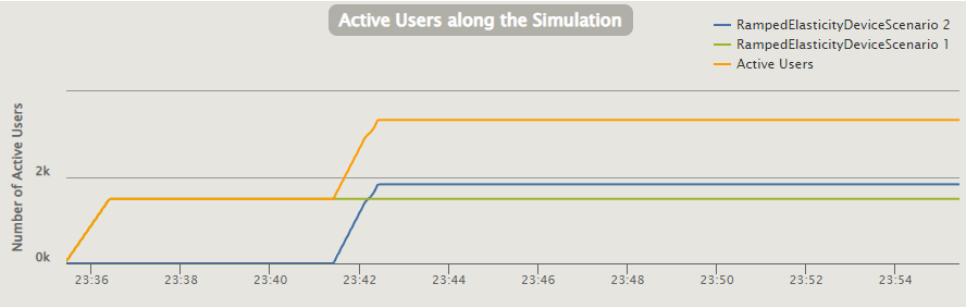

Figure 5.8: Gatling Load for the elasticity experiment.

In addition, we set the baseload to 1500 vessels, which is the number of vessels one node processed without problems in our scalability experiment. So under normal circumstances no scaling should be occur during the first 360 seconds. Then we have the first increment, which is no longer a multiple of the base load. From our scalability experiment we know that roughly 2000 vessels will over-stress one node. As described in the Process section, we need a low and medium scenario. Therefore, we decided to add 2000 vessels in the low scenario to provoke a scaling action. And for the medium scenario we would add 1500 vessels more, so 3500 in the increment step. That range of 3500 and 5000 vessels active should lead to one scaling action more, that means it should use two nodes for

the low and three for the medium scenario. Together with two scaling threshold values, the loose and tight utilization of Aslanpour et al. [ATTG21] threshold tuning experiment. That means a CPU utilization of 0.7 and 0.9.

Set-Up

The plan elasticity experiment is shown in the following Figure 5.9. The plan is to run two experiment runs back to back. After experiment run one is finished, we wait until the autoscaler scales back to one node. In addition, the utilization is steady below 0.05, which means the service is idle. And the last scaling decision was more than two minutes ago to avoid a delay of a scaling action. Last but not least, we delete the Pod of the Node Utilization Scaler to ensure no problem persists and ease the log recording.

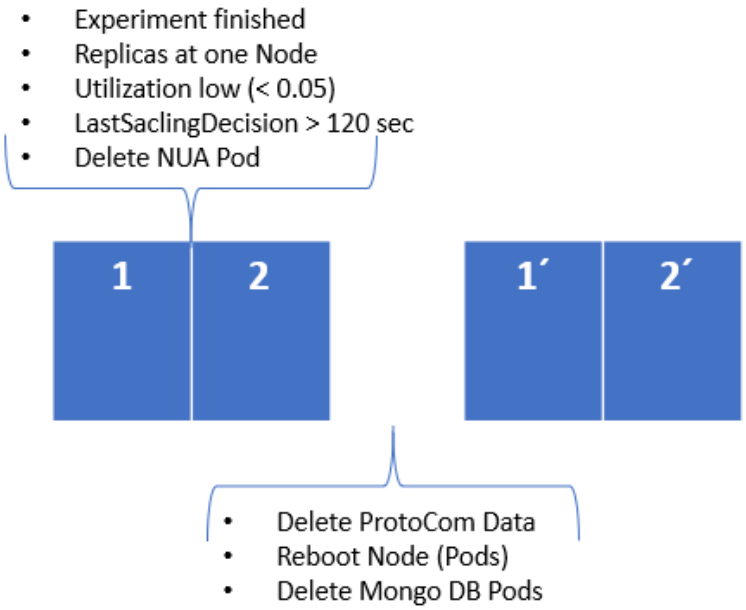

Figure 5.9: The process of the elasticity experiment execution.

Figure 5.10 shows the experiment scenarios with the two load scenarios, the tuned thresholds, and how often we repeated the experiment.

In addition, we recorded the response time through Gatling and the nodes used through the log of the node utilization autoscaler Pod. Other than in the scalability we do not exclude the ramp-up period, because the transient phase is of interest to judge about the scaling policy.

Load	Threshold	Executions
Low	0.7	5
	0.9	5
Medium	0.7	5
	0.9	5

Metric
Response Time
Nodes Used

Figure 5.10: The elasticity experiment scenarios and the collected metrics.

Elasticity Experiment Results

The system acted as we expected it, so it did not scale during the baseload phase. It scaled soon after the first increment step started, in every case to two nodes.

But only for the loose threshold of 0.7 CPU utilization, the scaler scaled to three nodes. The reason is with 5000 vessels, the utilization reached from time to time 0.8, but only once in of 14 runs (we added two more, each 5000 scenario, to gain confidence) it could have been enough for the tight threshold of 0.9. But that happened in one of the runs with 0.7 as a threshold. Therefore it changed nothing.

However, we cannot conclude that the utilization behaved differently because of the lower threshold. In terms of response time, the threshold made not a difference in the low scenario, compare Figure 5.11.

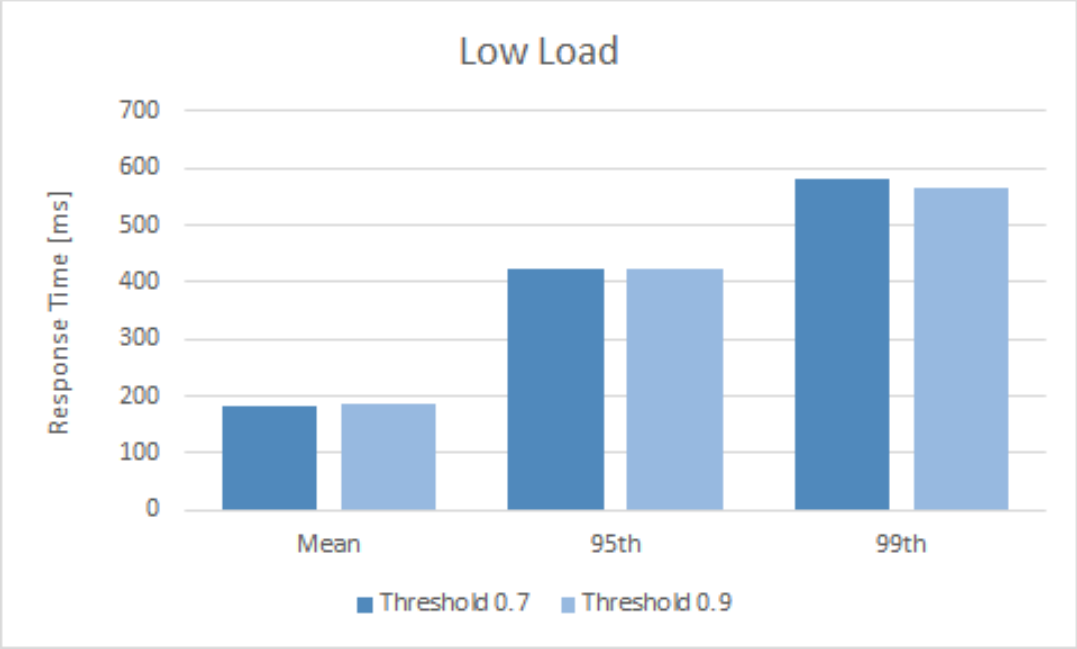

Figure 5.11: The threshold tuning experiment response time distribution - the low scenario.

As you would expect, the loose threshold of 0.7 leads to more resources used and has the faster response time. The lower threshold value is in the lead for both the mean and the percentiles. But that does not tell the whole story, in some runs the higher value was competitive without scaling to a third node. We also can observe a higher standard deviation for the that threshold, we reflects that the system behaves less constant and it depends on chance how good the response times are during the transient phase. In addition, both scaling policies keep the system at our SLO of 1000 milliseconds response time. But for the 99th percentile both fail to keep the SLO with the medium load.

Another thing to note is that we still experienced peaks during the ramp-up period, even if the period was much higher than at the scalability experience. What is also visible is a dent or plunge during the ramp-up of the increment step, as shown in Figure 5.13 or Figure 5.8. The cause of the behavior

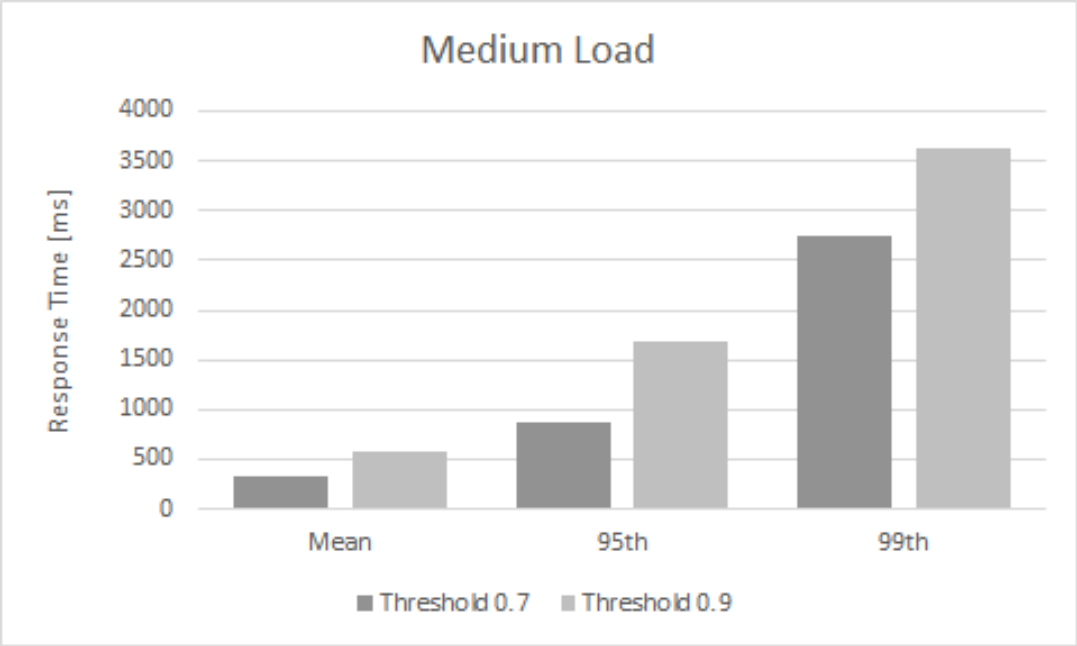

Figure 5.12: The threshold tuning experiment response time distribution - the medium scenario.

is timed-out invocations of setOnline. It was only a little fraction of devices, at most 400 from 5000. That was consistent through all runs with the same amount of vessels. Therefore, we do not think that it distorted the result.

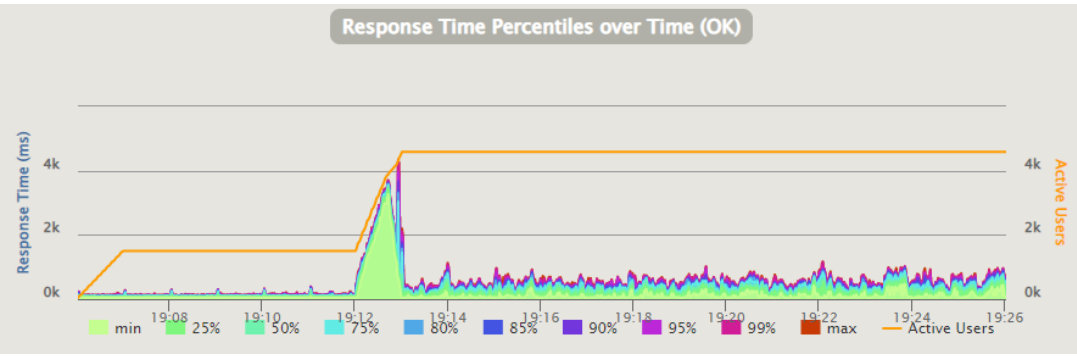

Figure 5.13: Response time peaks during the elasticity experiment.

Another thing to note is visible in Figure 5.14, where we see the number of requests per second. We would expect that Gatling distributes the device sending data equally. But there is an oscillation that gets bigger with more devices. And has peaks and valleys after some setOnline invocations timed out.

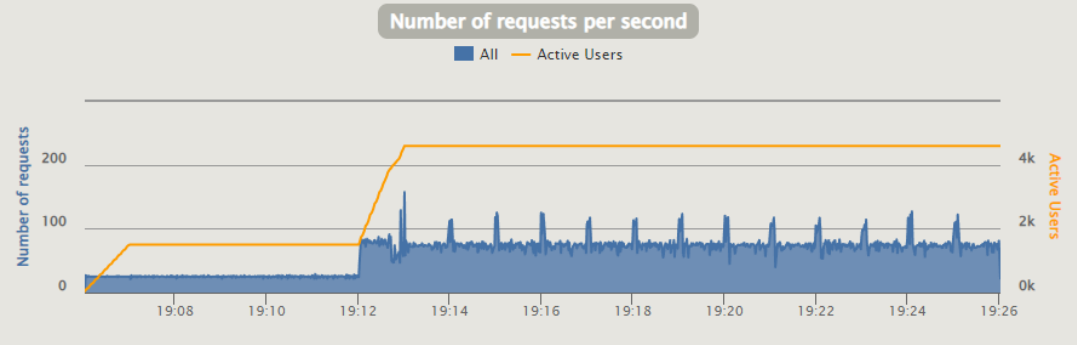

Figure 5.14: Request distribution of the elasticity experiment

5.3 Model Results

A model of the MoSaIC system already existed, and it was updated constantly by the project partners. But the model never consisted of the full demonstrator, which was also still under construction during this thesis. However, we used it as inspiration and modeled our version because it was impossible to use the model for our profile. But it still represented the state of the actual development.

The concept that we aimed to realize was to include a load balancer, which changes the probability of the transitions during the simulation, see Figure 5.15.

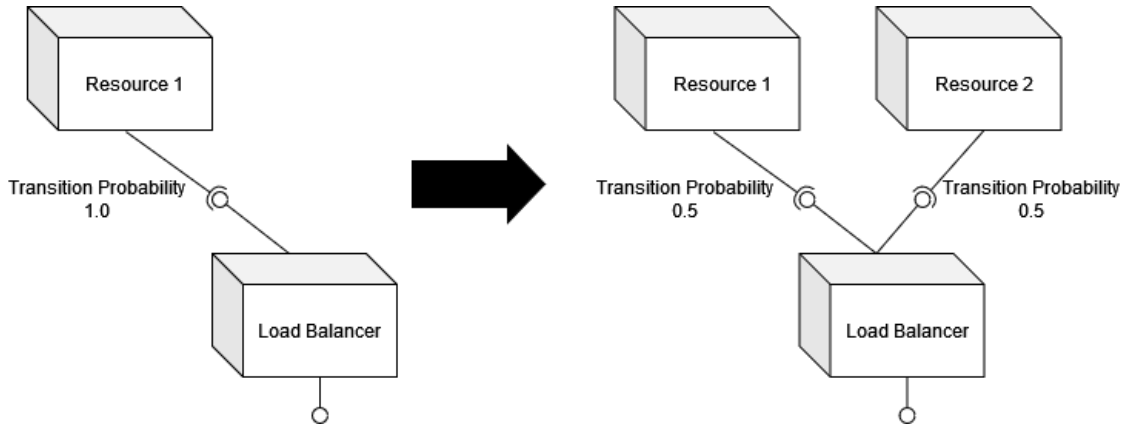

Figure 5.15: The concept behind our implementation.

In the next section, we describe the model superficially when it is part of the demonstrator and is part of the MoSaIC project. Whereas the parts we modeled for our purpose, we describe it in more detail.

Repository

In the repository model, the Mongo database, the RabbitMQ Broker, and each of the three MoSaIC demonstrator services have an interface and component. The three services are Device Communication, Data Provider, and Data Processing. As described in the foundations, the corresponding interfaces provide the required and implemented methods for the components. For instance, each of the three services requires the interface of the Mongo database and the broker, but none needs the mutual interfaces of the other demonstrator service. As described, the communication between them works over messaging implemented via the Broker component. The RabbitMQ Broker is different because it has three interfaces, and these three are infrastructure interfaces, compare Figure 5.16.

Device Communication, Data Provider and Data Processing required only the message queue or infrastructure interface it needs. The device communication only needs the data available queue, the Data Provider the reply and request queue, and the Data Processing all three. The component has for each queue a Passive Resource which is used in the SEFF.

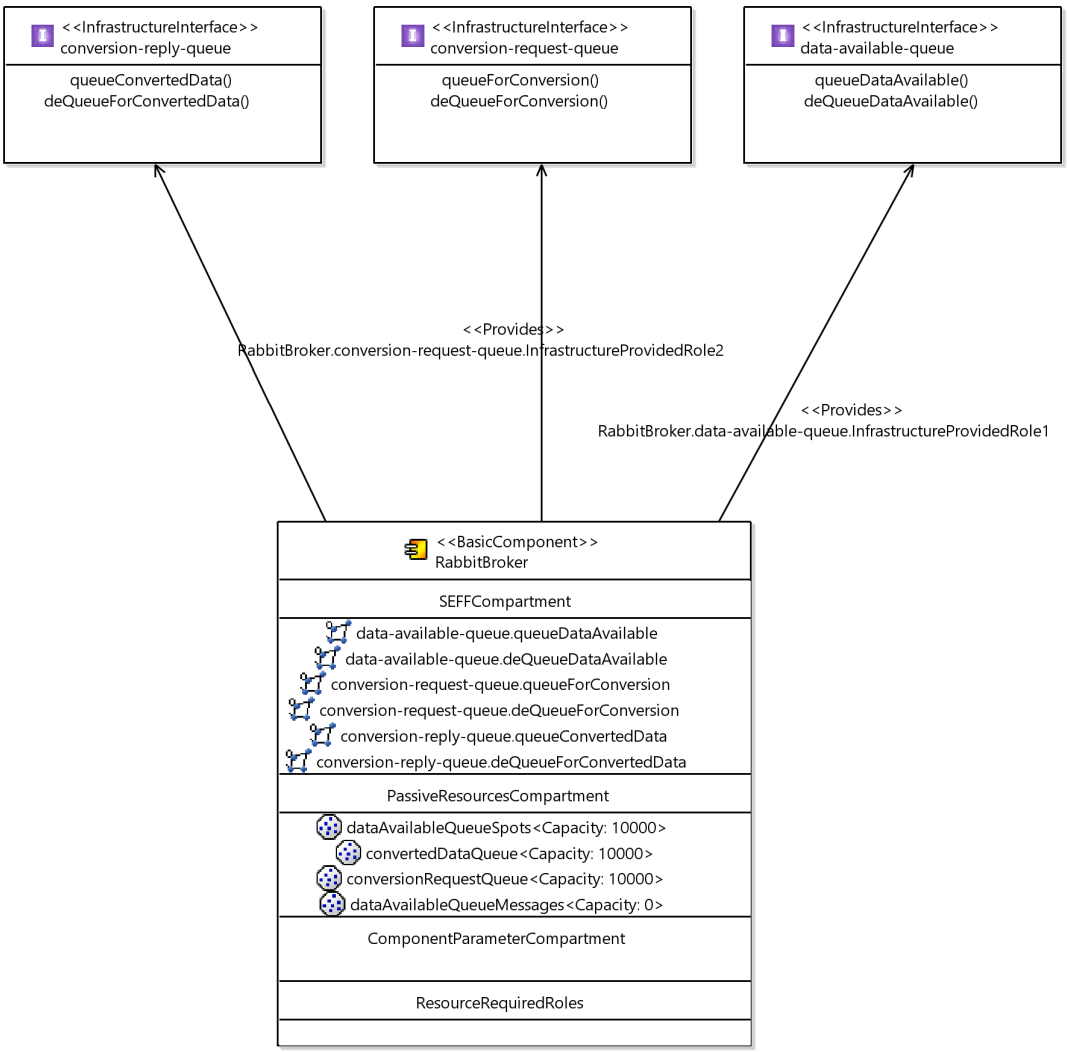

Figure 5.16: Picture detail of the Rabbit Broker with the corresponding interfaces.

For the scaling functionality described above in the concept, we need an additional component and interface. See Figure 5.17 for that Load Balancer. In the graphic, the Load Balancer provides an interface for Device Communication. But at the same time, it requires the interface x times.

The idea of the load balancer is to provide the interface of the component to scale once and require it x times, where x is the number of the maximum number of replicas to scale. In our case, for the node scenario, the number is four. Analog as in Figure 5.17, Data Provider and Data Processing respectively, the provided and required roles for the interface are similar to the Device Communication. That also means that we model another SEFF in the Load Balancer for each of their methods.

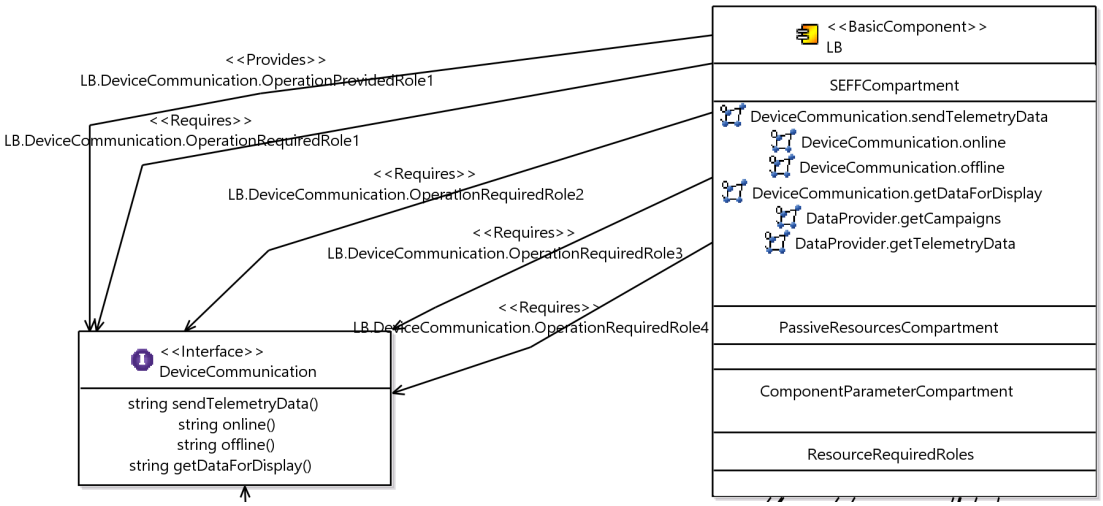

Figure 5.17: The load balancer component and its interface.

SEFF

The SEFF of the Device Communication, Data Provider, and Data Processing, the Mongo database are generic. For instance, *sendTelemetryData* compare Figure 2.5 from the Device Communication has an internal action with a CPU resource demand and two external call actions afterward that call the database to store and data available queue of the Broker. The Data Processing is slightly more sophisticated, as here we have the case where the data is either in the cache or not. Here, a probabilistic branch action has two transitions with a probability of 0.5. So that in 50 % of the cases, we model a resource demand for loading the data in the cache. For the other 50 % nothing is modeled because then the data is already in the cache.

As the Broker, the infrastructure interfaces have a different model because they model the behavior of a message queue. So within the SEFF, a token of the Passive Resource capacity is acquired and then released again, see Figure 5.18, below you can see the four different Passive Resources.

For the Load Balancer, the idea in the SEFF is to create a branch action with probabilistic branches. As described in the concept above, one branch starts with a 1.0 probability and the other ones with 0.0 probability. In the initial state, there is no scaling or load balancing.

The SEFFs of the Load Balancer component all look the same. We do not model a new behavior. Instead, we externally call the original SEFF. That instantiates the expected behavior of a load balancer that passes the load without changing the functionality. We do that x and four times in the probabilistic branch action respectively. It is crucial to edit the role external service in the model explorer (properties view). Otherwise, the load balancer has no effect because each branch points to the same Assembly Context. But more on that in the Assembly section.

Figure 5.19 shows an example of how that SEFF looks, in that case we only would scale to three nodes. However the concept stays the same, only one transition has a probability greater than zero. Our scaler scales during runtime, we describe the realization of our concept in the Self-Adaption Rule section. Here, we transform the probability value, which equally balances the load between the scaled instances.

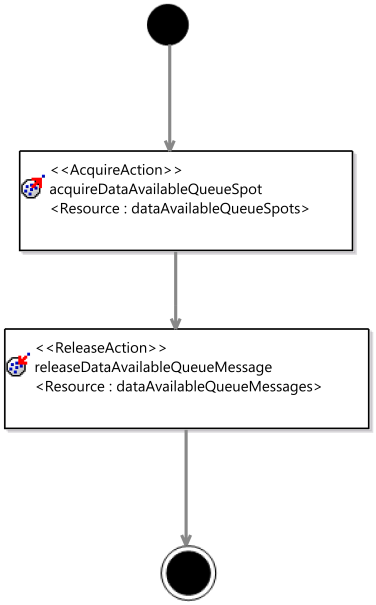

Figure 5.18: The SEFF of the Infrastructure interfaces.

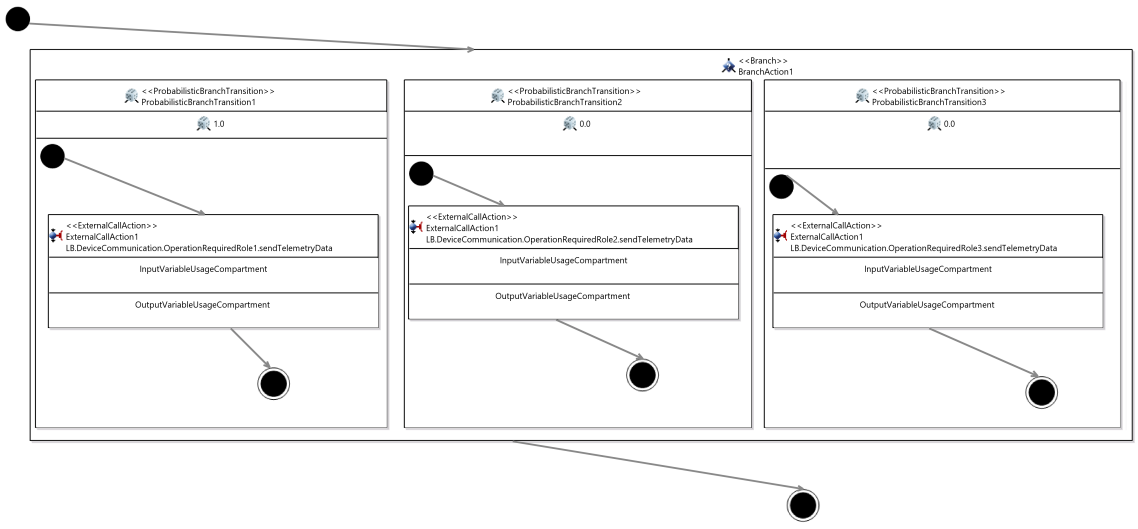

Figure 5.19: The SEFF of the Probabilistic Branch Action.

Assembly

The system or assembly model of the MoSaIC demonstrator system starts with four interfaces from the outside. Those interfaces are the REST interface for the users and the Ingress for the Device Data. The REST interface subsequently connects with the Data Provider and the Ingress with device communication. All services communicate over queues, resulting in each Assembly Context of the three main components requiring infrastructure roles to connect to the Broker context. However, we cannot do that with the Data Processing and the message queues it uses. The message queues have their own interface and subsequently their own Usage Scenario. We will explain the latter later, however, this means we need to remove the Data Processing from the Load Balancer. The consequence is, each Data Processing Assembly Context replica has its own message queue interface and is not connected to the load balancer.

We add the load balancer in front of the Data Provider and Device Communication components and direct from there to the actual instances, to visualize that see Figure 5.20. Here you can see all the required interfaces that we modeled in the repository model. At this point, it is crucial that in the SEFF, each external call points to a different required Role because we replicate the demonstrator Assembly Contexts x times. In the node scenario, we now have four Assembly Contexts per service. Respectively twelve overall instead of three. We recommend naming the Assembly Contexts with an ascending number, e.g., Data Provider 4.

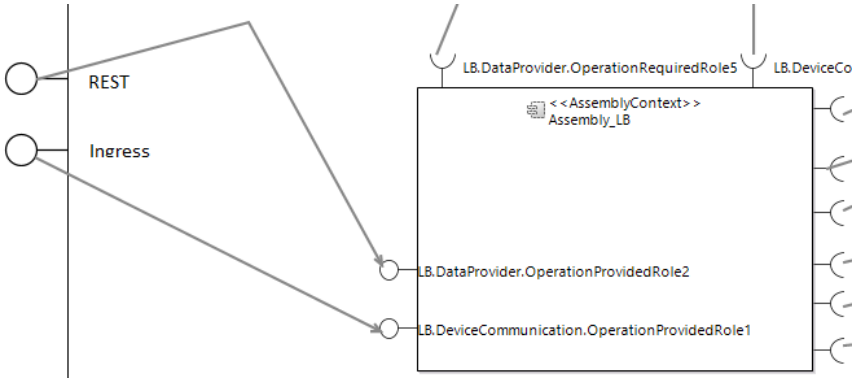

Figure 5.20: Picture detail of Assembly Context of the load balancer.

In addition, only the components to scale, the demonstrator services, get a replicate as Assembly Context. We do not scale the Broker or the database. That all means the components to scale call the same database or Broker interfaces as before.

Resource Environment & Allocation

The initial model has three resource containers with a CPU Processing Resource, one for the database, one for the broker, and one shared between the Device Communication, Data Provider, and Data Processing. The Allocation Assembly Context is placed in the allocation model. The containers were all linked with each other through a linking resource.

For our load balancing approach, we add three additional containers with the same properties. That includes the processing rate and replicas. Then we connect them with the existing linking resource, too. The allocation depends on the implemented scaling mechanism. For our homogeneous node utilization scaler, we replicate the existing allocation on the new resource container. For the inhomogeneous scenario, this has to be changed as well for the HPA. The HPA regardless, if the subsystem capability is used or not, requires more adaption. For instance, if the HPA only scales one service, only that has to be replicated.

A minor implementation detail, the load balancer has an Allocation Assembly Context, too. It is needed to allocate on a resource container, like every other context in use. But it does not generate load. Therefore we can neglect it and put it on any available resource container.

Usage

The Usage Model has three different Usage scenarios, one for the devices and two for the messaging of the broker. While the latter is just an external call, the model of the device is the process for setting the device online, sending telemetry data, and setting it offline. The parameter for the devices in a closed workload is one user with a think time of ten seconds. The two messaging scenarios it is also a population of one and a think time of ten seconds.

Figure 5.21 shows the Usage Scenario of the vessel. We added a ten-second delay after sending data, which is in a loop with 100 iterations. That is set to 100 to emulate the forever.

It is also possible to model the behavior in an open workload, where instead of the number of users and the think time, the interarrival rate becomes specified. This rate could be calibrated with values from measurements to get better accuracy. However, a problem here is that modeling it as shown in Figure 5.21 executes setOnline as often as sendData, which is wrong. The price of this solution is to neglect the setOnline call. Figure 5.22 shows the new Usage Scenario for send data.

The Usage Model includes additionally to the send Usage Scenario x Usage Scenarios for the data available message queue, which manages the communication with the Data Processing component. Figure 5.23 depicts four Usage Scenarios, x is four because the system has four nodes where the Data Processing can run on. The population is set to 1 to activate the message queue, that means only if all four nodes are used each Usage Scenario has a population of four. The case shown in Figure 5.23 shows one activated message queue.

However, as this concept made sense for an equal distributed load, with the behavior seen above during the elasticity experiment, it became questionable to model the usage like that. To address that source of inaccuracy, we decided to change from an closed workload to an open. In the open workload we just simulated the the data sending by an specified inter-arrival time.

We saw problems with using the gamma distribution in the Usage Model. Firstly, it was not documented how to specify gamma (shape or rate first) and we experienced problems. We obtained only a few data points in the sendData case compared to the exponential distribution. Therefore we deiced to use the exponential distribution to avoid these problems.

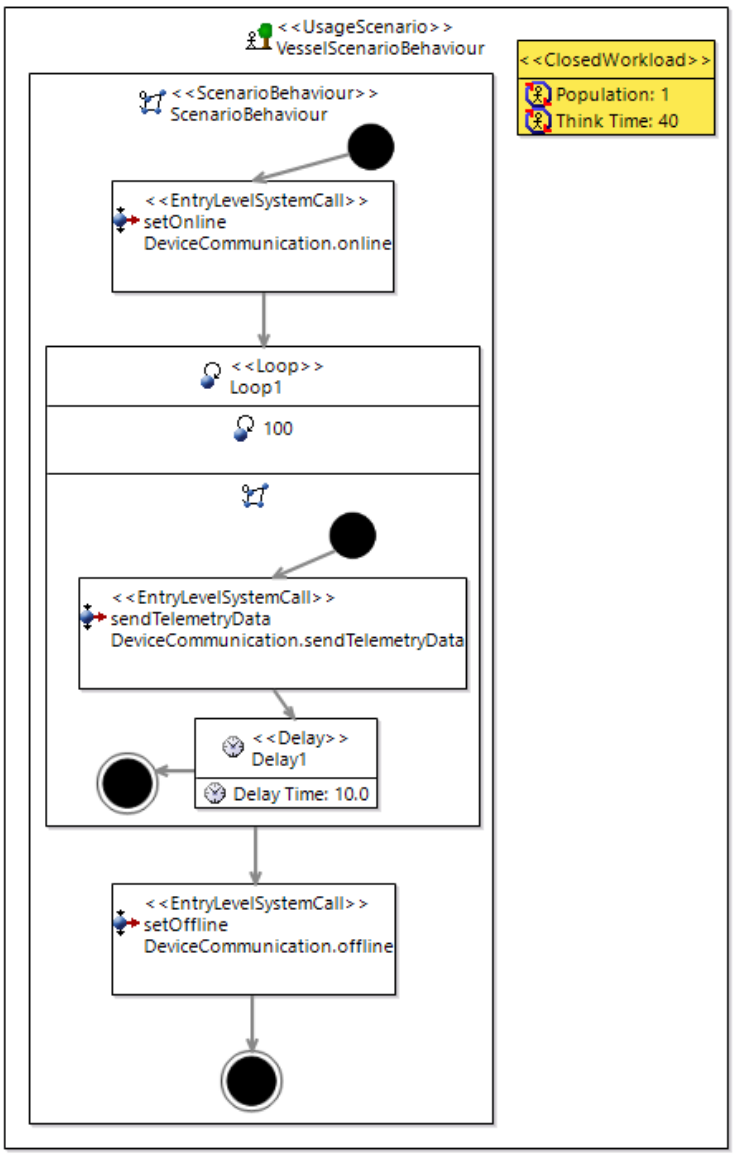

Figure 5.21: Experiment Usage Scenario with a closed workload.

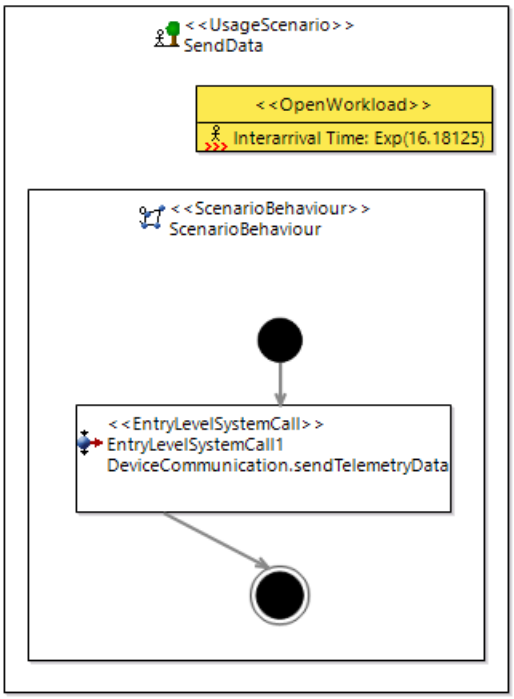

Figure 5.22: Experiment Usage Scenario with an open workload.

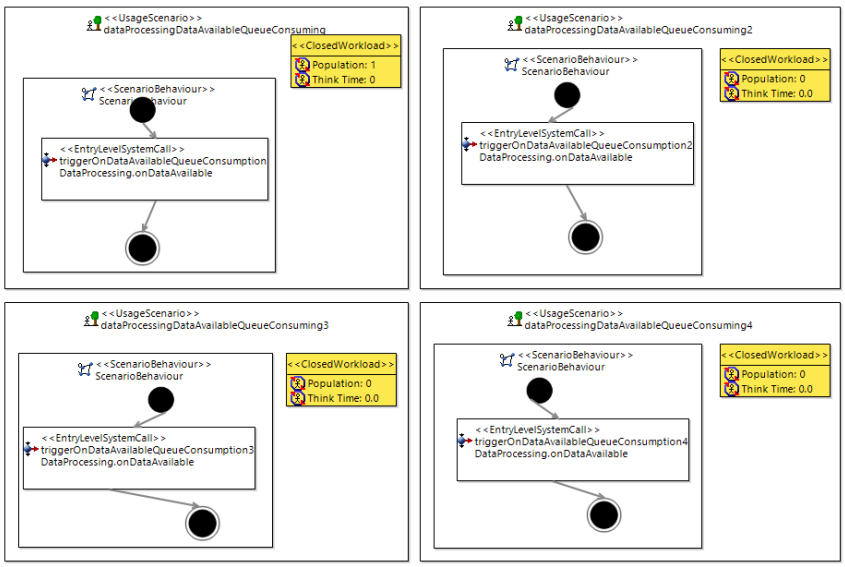

Figure 5.23: Usage Scenarios for the data available message queue.

Usage Evolution

The Usage Evolution consists of the Usage Model and the active user (vessels) evolution over time. In the latter, we imitate the load we issue with Gatling and save it in a DLIM file.

To model our load, the custom editor was not enough. We had to use the tree editor. We specified the rough behavior, the number of peaks, and that the trend type. The options noise/burst or overarching trends are not used. Then we edited the DLIM file in the three editors. Theoretically, it is possible to model the load solely in the three editors, as you can add as many time-dependent function containers as you need and specify their properties.

As described above, a DLIM file has a trend type. In this case, we decided on a logarithmic trend with order ten. We use it because it is the easiest way to imitate the steps. Other types as the polynomial or sinus are not suited to do that. However, a linear trend allows a specification of such steps, but we decided against them. The reason is that with a linear trend, all devices become active at the same time. To our understanding, this is not how Gatling works, which leads to a logarithmic trend with high order.

We had to adapt the Usage Evolution after the Usage Model adaption from a closed to an open workload. We tried to model the evolution of the inter-arrival time, compare Figure 5.24. However, this evolution caused problems and did not work as expected. Therefore, we decided to omit the Usage Evolution for our simulation model.

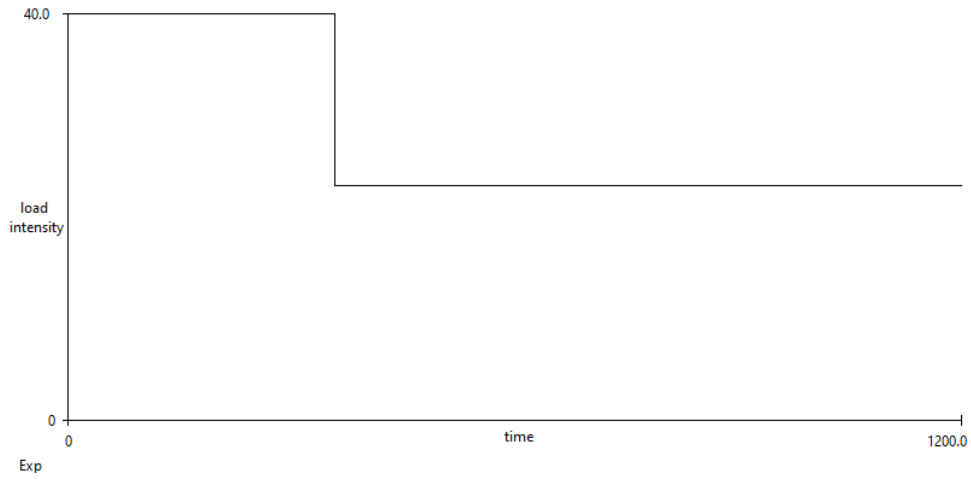

Figure 5.24: The usage evolution of an decreasing inter-arrival time.

Measuring Point & Monitor Repository

As described in the foundation, the Monitor Repository and Measuring Points allow controlling what usage or resource is measured and which metric is monitored. We first create the Measuring Point for the Resource Containers of the nodes we want to scale. In the next step, we created the Monitor Repository by creating one Monitor per Measuring Point and multiple monitor specifications per Monitor. That means we collect at least three different metrics per Resource Container, Utilization, State of Active Resources, and Description Resource Demand. Depending on the metric, the

specification has a further child, which specifies how the metric is collected. That child can be feed-through or time-driven. The first does not specify anything in addition, but the time-driven allows changing the window increment and length. The length refers to the initial point in time to record the measurement, while increment refers to the time between the consecutive measurements. Further, the utilization of active resources is a special case. It does not work without a monitor specification of active resources within the same Monitor to be measured.

You can find a more detailed description of modeling measuring points and monitor repositories here [3], point 13. Another possibility to specify the measuring point and monitor repository is the SimuLizar Usability Extension [NMM+18]. The extension supports the creation via wizards and saves work by creating some parts automatically. This extension is helpful but has some bugs. After saving, the dashboard flickers. Moreover, using Palladio version 4.3, the existing Measuring Point and Monitor Repository are deleted and replaced.

In addition, we faced the problem that the utilization of active resources metric only worked with 4.3 or lower. With 5.0 or nightly, we experienced a NullPointer Exception, which seemed a bug to us.

MDSD Profiles and Stereotypes

At this point, we wanted to introduce the self-adaption rules. The QVTo scripts have a major drawback. It is impossible to store values during the execution, which makes the scaling policy stateless. For concepts as the quiescence period, this is a show stopper. However, with MDSD profiles and stereotypes [4] it is possible to use the so-called taggedValues. In these taggedValues, you can store before, during, and afterward, values that enable stateful concepts as the silent, quiescence period or how many instances are scaled right now.

Therefore we created a scaling policy EMF profile [5], which is extended by the MDSD profiles. We imported the metamodel element system because our tagged values should appear in the assembly respectively system model, compare Figure 5.25. When the upper element appeared, we created the stereotype ScalingPolicySystems, which extends the system. Then we added two tagged values to the stereotype, scaledUnits, and lastScalingDecision. Both are EDouble fields with the default values zero. That creates the capability to have stateful scaling policies specified through QVTo.

This solution is simple, but it is cumbersome to execute. Apart from the profile, you need to specify an architectural template. Otherwise, we were not able to test it. With that template ([6]) and execute the *.architecturaltemplates* file in an inner- eclipse application, you can apply the profile/stereotype in the model and also use it with the QVTo script.

This solution has pitfalls because exception messages like “no profile applied” or “invalid/out of bounds” appear. It is not wise to close the model before removing the profile or have two models with the same profile applied within one workspace. That caused trouble because the check

[3] https://sdqweb.ipd.kit.edu/wiki/Palladio_Usecase_Tutorial
[4] https://sdqweb.ipd.kit.edu/wiki/MDSDProfiles
[5] https://sdqweb.ipd.kit.edu/wiki/EMF_Profile_Definition
[6] https://sdqweb.ipd.kit.edu/wiki/Architectural_Templates

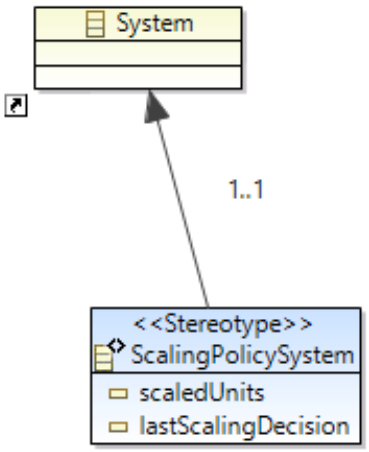

Figure 5.25: Our EMF Profile diagram

hasProfileApplied fails. It only works when the profile/stereotype is applied to a model/workspace where none is already applied. The solution for us was to do our simulation, and before the closing eclipse, we removed the profile.

In addition, an error during the execution of the QVTo to self-adaption requires a restart of the eclipse. An example of that is the state when values from the *taggedValues* are invalid. That is an indication that something is wrong, and you need to restart. Also, we could not use Integer fields respectively using *setTaggedValue*. Here the warning that the call is ambiguous was prompted by the editor. We fixed that by changing from Integer to Double. Apart from that, it seemed that there were some Palladio version conflicts. In our case, a model created with the nightly version did not work in 4.3, even after replacing the representation.aird to solve the Sirius version conflict. To ease that in the future, we wrote a tutorial to use our profile.

At this point, the question arises, why do we not directly implement an AT when we defined one. We looked at it and had no solution for the problem that we could not use in *prm:PRM* (Palladio Runtime Measurement) within the AT completion. If we did add that and execute the simulation, our AT was unapplied. We need the Runtime Measurements because the silent period requires tracking the simulation time at the point in time a scaling action becomes executed. There is a solution for that, but because of time reasons, we continued with the profile.

Self-adaption rules

A self-adaption rule works as a model transformation during runtime. In our case, as described in the concept above, to transform the probabilistic branch transitions. But if and only if the scaling conditions are met, e.g., the threshold is reached, no scaling operation was done for the duration of the quiescence period, and scale the right amount to the right time. It is possible to create the described load balancer from above during runtime, or you rely on elements that already exist in the model. That has the consequence of a precondition that, e.g., resource containers already exist

before executing the scaling policy QVTo script. That is easier for usability but also harder to implement and to understand. So as described above, we modeled the elements with the Sirius editors.

Within the QVTo code, the first step is to set the properties according to the scaling policy to model, see listing 5.1.

Listing 5.1 Set the properties.

```
    // The utilization thresholds
    property upperThreshold: Real = 0.9;
    property lowerThreshold: Real = 0.3;

    // constraint
    property quiescencePeriod : EDouble = 120;
```

We then start with the preparation of the scaling. That means collecting the runtime measurements, allocation, and system elements from the executed model. In addition, it is crucial too to check if the stereotype/profile is correctly applied. In the next step, we ensure that the allocation and measurements are not empty.

After that, we can start with the scaling process by collecting the measuring point of our selected metric by its id and checking it against the defined thresholds. The entry call is shown in the listing 5.2, where we call the *checkCondition* method with the selected Measuring Point of the Resource Container and the utilization metric.

Listing 5.2 Entry call of the *checkCondition* method.

```
  // ID from the measurement specification
  property overallUtilizationId: String = '\_Z-x\_oDdwEeysRuXy5LSnPA';

  Set {prms->any(measurementSpecification.id=(overallUtilizationId)))} -> xcollect(
runtimeMeasurement.checkCondition());
```

In the *checkCondition* method, we first retrieve the values from the tagged values. At the initial entering, the number of scaled units and the last scaling decision is zero. Then we move forward to check the measurement value and if it is reached or not. We do that as shown in listing 5.3 by checking against the upper threshold in case of scaling out. Further, we check if the current simulation time is greater than the point in time of the last scaling decision plus the silent or quiescence period. In addition, it is also possible to add a further condition, e.g., that in the first 100 seconds of the simulation, no scaling action is taken. In addition, if the simulation time is smaller than the quiescence period, the constraint will prevent any scaling action. We needed to check that to avoid an incorrectly implemented constraint. If the scaled units are smaller than one - no scaling action happened before - the constraint does not apply.

Listing 5.3 Here the threshold and constraints are checked.

```
  if(self.measuringValue > upperThreshold and ((simulationTime < quiescencePeriod and
ScaledUnits < 1) or (simulationTime > lastScalingDecision + quiescencePeriod))
```

When the threshold and the constraints are evaluated as true, we need to know the scaling history. Scaling for the first time is not difficult because we know that the scaled units are zero, and we add the first node replica. But the process is different when there already were scaling actions executed. Then we need to check that beforehand. We do that with a switch case of the current scaled units, see listing 5.4. Then we call the scaling action and update our tagged value last scaling decision with the current simulation time so that the constraint from above is evaluated correctly.

Listing 5.4 The process how to determine, which scaling action to take.

```
    switch {
      case (currentScaledUnits = 0) do {
        scaleOutModelByOneUnit(allocation);
        setTaggedValue(systemAllocation, simulationTime, spStereotype,
lastScalingDecisionTagged);
        };
        case (currentScaledUnits = 1) do {
        scaleOutToUnit2(allocation);
        setTaggedValue(systemAllocation, simulationTime, spStereotype,
lastScalingDecisionTagged);
      };
    }
```

Let us assume *currentScaledUnits* is zero. Then we call the function *scaleOutModelByOneUnit* with the allocation contexts of the model as a parameter. What happens next is that for each of these allocation contexts, we filter for our load balancer, see listing 5.5. As we named it LB, we can find it with its entity name and then go deeper into its SEEF and search for probabilistic branch transitions. In our case, it was the easiest to identify them by their entity name. The precondition for it is a consistent entity naming.

In the case of scaling the first node, we only need to transform two branch probabilities, the one that had 1.0 and one which was zero, to 0.5 for both. As a result, we now balanced the load to a second instance during runtime. For more instances, we set the probability to $1/x$, where is the number of replicas.

In addition to the branch probabilities, we need to change the population value of the Usage Scenarios as well. This is works analogously as above, compare listing 5.6.

Then the only thing left is to update the current scaled units tagged value, in this case after the first scaling action, to one.

Listing 5.5 The scaling functionality.

```
allocationContexts -> forEach(allocation) {
  switch {
    case((Assembly_LB)) do {
      forEach(allocation.seff) {
        switch {
          case(seff) do {
            var branchSet:Set(ProbabilisticBranchTransition) := xselect(seff::
ProbabilisticBranchTransition);
            branchSet -> forEach(probabilityBranchTransition) {
              switch {
                case((ProbabilisticBranchTransition2))) do {
                  probabilityBranchTransition.branchProbability := 0.5
                };
                else switch {
                  case(ProbabilisticBranchTransition1))) do {
                    probabilityBranchTransition.branchProbability := 0.5
                  };
                };
              }
            }
          };
        };
      }
    }
  };
}
```

The only difference for scaling more instances is how many probabilities are changed and setting the right number of probabilities. The process to scale in is done analogously, only by adopting the threshold check and calling different methods to transform the branch transition probabilities and reducing the *scaledUnits*.

The scaling mechanism we showed above referred to the simple Node Utilization Scaler. However, it is also possible to model the more complex HPA scaler. First, we need to replicate the HPA formula see the listing 2.1 in the foundations. The difference is that we calculate the ratio beforehand by dividing the measurement through the thresholds and multiplying that with the current number of replicas or units, see listing 5.8.

In the further process, we check the ratio and then the scaling status of the system to determine the scaling actions to take. So let us assume the ratio is below one because the measured value is smaller than the threshold. That will never result in a scale-out action only to scale-in. The next question is, how many Pods are running. When we know that, we can scale down to one Pod. See listing 5.9, for different ratios the approach is analog. Although a ratio greater than one can also result in scaling out. For instance, a ratio greater than two and smaller than three results in scaling one instance if only one node is running. The reason is the ceiling, which we implemented without rounding but using multiple if and elif branches.

Listing 5.6 The scaling functionality.

```
 usageScenarios -> forEach(usageScenario_UsageModel){switch {

     case(usageScenario_UsageModel.entityName=('dataProcessingDataAvailableQueueConsuming2'))
do{

       if(usageScenario._population = 0){log("Second Queue added");
         usageScenario._population := 1;}

     };

   };


 };
```

Listing 5.7 Updating the scaled units *taggedValue*.

```
  var newNumberOfReplicas:Real := 1;
  setDoubleTaggedValue(systemAllocation, newNumberOfReplicas, spStereotype,
scaledUnitsTaggedValue);
```

The difference to the Node Utilization Scaler is that we call more than one method to change the branch transition at a time. That means the HPA can add more resources at once and is faster if the load increases fast.

We hard-coded the scaling process for demonstration, but it is possible to do that dynamically. For instance, we could introduce another *taggedValue* called nodes to scale, which has to be specified in the profile before the simulation. As a result, the script creates the whole load balancer in the QVTo script.

Listing 5.8 The HPA formula in realized in QVTo.

```
    var ratio: EDouble  :=  (self.measuringValue) / (threshold) * currentUnits;
```

Listing 5.9 The HPA scaling modeled in QVTo.

```
    if (simulationTime > lastScalingDecision + quiescencePeriod or currentScaledUnits < 1){

      if (ratio < 1){


      if(currentUnits = 4){
          scaleInModelAtOneUnit(allocation);
          scaleInToUnit2(allocation);
          scaleInToUnit3(allocation);
          setTaggedValue(systemAllocation, simulationTime, spStereotype,
lastScalingDecisionTagged);
        }
      elif(currentUnits = 3){
          scaleInModelAtOneUnit(allocation);
          scaleInToUnit2(allocation);
          setTaggedValue(systemAllocation, simulationTime, spStereotype,
lastScalingDecisionTagged);
        }
      elif(currentUnits = 2){
          scaleInModelAtOneUnit(allocation);
          setTaggedValue(systemAllocation, simulationTime, spStereotype,
lastScalingDecisionTagged);
        }
      else{

        }

    };
```

Listing 5.10 ProtoCom demand.

```
resourceDemand.initializeStrategy(DegreeOfAccuracyEnum.HIGH, 1000);
```

5.3.1 Calibration

The calibration process is twofold. Firstly, the calibration of the ProtoCom performance prototype to create a demand that corresponds to the demand of our model. ProtoCom simplifies that by allowing to set a processing rate, see 5.10.

In the implementation of the device communication a CPU demand, *protocomService.cpuDemand(50)*, of 50 is issued for send data.

At this point, we have to decide either to use the 50 ms in the Resource Demand or the exact time the 50 ms needs to run in the cluster. The design decision was in favor of using the 50 ms and introducing a possible source for inaccuracy.

During the experiment, it became clear that a recalibration of ProtoCom before creating the demand respectively a restart of the nodes reduces this inaccuracy. This observation was backed by running several benchmarks. See Figure 5.26, where a 1000 load was issued and the result show a wide range, except for minion 06, which was restarted just before running the benchmark.

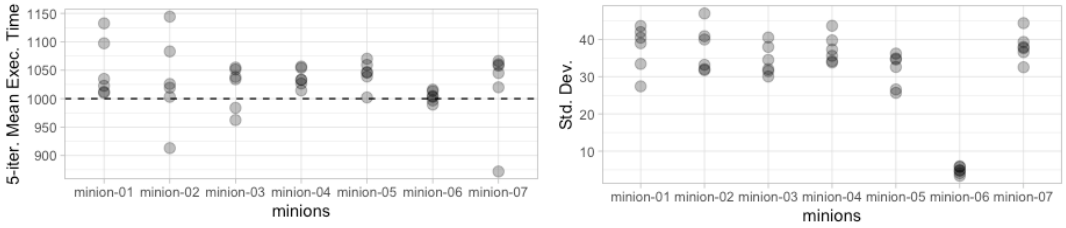

Figure 5.26: Result of a benchmark run on all available nodes.

The structure of the BW cloud and the virtualized nodes/the virtual machines allow no control on the underlying hardware. We guess that could cause such mismatches respectively require a recalibration before the actual experiment. Therefore each resource container in the model has a processing rate of 1000.

The second part of the calibration is to calibrate the elements of the system that trigger now ProtoCom demand. The *setOnline* method does need as much time as it takes to process the invoking. Therefore we run a benchmark load-test and use the mean duration for the Resource Demand.

Apart from the ProtoCom demands, the system has elements that do not use ProtoCom. The *setOnline* method does need as much time as it takes to process the invoking. Therefore we run a benchmark load-test and use the mean duration for the Resource Demand. That became obsolete after we removed the method from the Usage Model.

We added another calibration part with the decision in favor of the inter-arrival time. We used the R package fitdistrplus from Delignette-Muller and Dutang [DD15] to create the distribution rates that fit the inter-arrival time of the three workload variants. The reason that we only used three is to save

time. In addition, the difference between the different thresholds within a load was marginal. Before creating the Maximum Likelihood Estimation (MLE) of our distribution, we removed the zeros and outliers from the data set. The outliers are everything above the 75th percentile. The reason for removing the zeros is to allow the gamma fitting, which failed otherwise. The reason to remove the outliers is to compensate for the Gatlings inequal inter-arrival distribution. This inequality leads to the oddity that the inter-arrival time is decreasing with more vessels instead of increasing. It makes no sense to have a less intensive load with more devices. Therefore we opted to remove the outliers. After that, the inter-arrival rate increased with more devices. We choose the gamma and exponential distribution based on the histogram and to have a reference.

Figure 5.27 and Figure 5.28 visualize the fitting. Visually it seems that the gamma distribution fits better to the baseload, whereas for the medium and low load the exponential

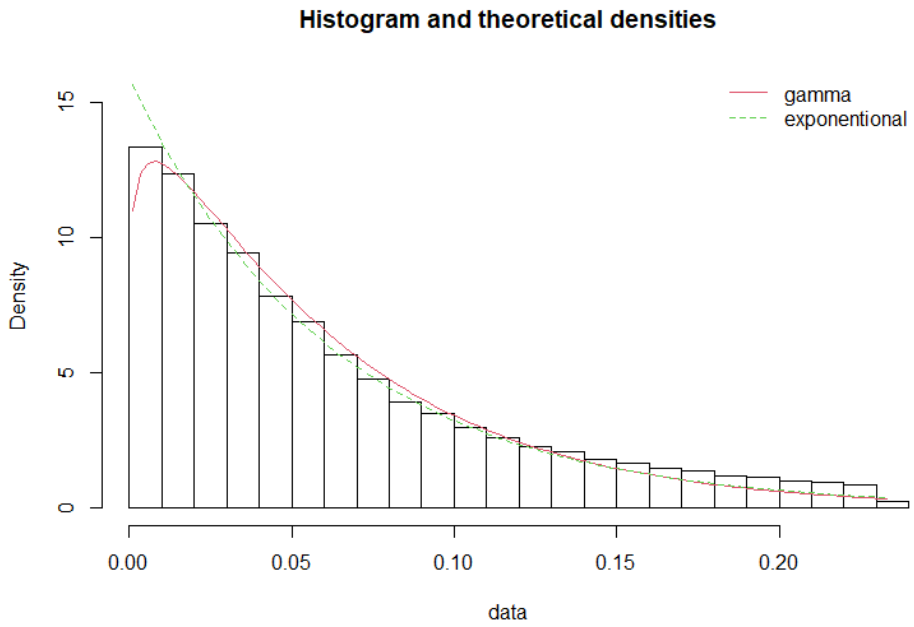

Figure 5.27: Comparison of gamma and exponential fitting for the low load.

The log-likelihood values supported our decisions regarding the baseload. The measure shows how well the maximum likelihood fittings fit the dataset. However, according to that measure gamma fitted the medium and low load better too. Although, the difference between gamma and exponential concerning the log-likelihood is not big. We assume the difference should not be severe and is not a major issue to only use the exponential distribution because of issues with the Usage Model. However, this is a subject for further evaluation.

The used distributions are, a rate of 21.07226 (SD 0.1005747) for the base load, a rate of 11.18811 (SD 0.01932798) for the medium load and a rate of 15.88159 (SD 0.03546713) for the low load.

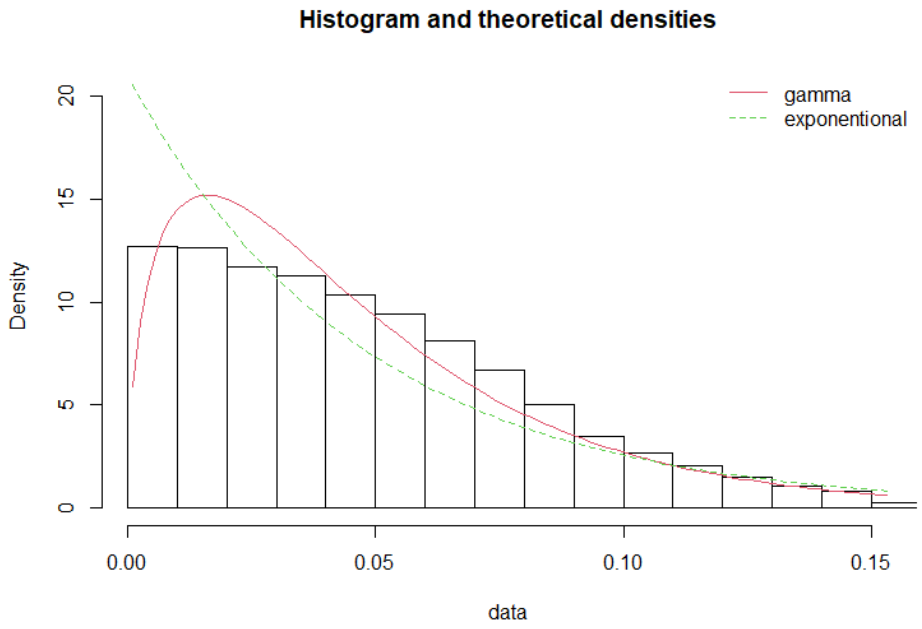

Figure 5.28: Comparison of gamma and exponential fitting for the base load.

5.4 Simulation Results

We run the simulations 500 seconds in the baseload with our QVTo autoscaler implementation switched on. We opted for a static system during the additional load phase because we wanted to emulate the scaling behavior. In our experiments, during the baseload phase scaling occurred, while during the additional load it did rarely occur. It only occurred in one of our four scenarios. The simulation used four nodes to 100 % during the whole simulation. We run the simulations with SimuLizar for 500 seconds and repeated them 10 times for each setting. The showed results compare the autoscaler QVTo to implementation that starts with four nodes.

The results are the following. The autoscaler implementation produces similar response time values, except for the higher percentiles. This fact is not surprising because of the scaling we have a transient phase, which results in a few higher response times. Figure 5.29 shows that behavior, with a big difference for the 99th percentile, with the difference is getting smaller with more data points. This observation holds true for all three different load scenarios, but the difference gets smaller with a higher load, and the response times are getting better, compare Figure 5.30 and Figure 5.31.

Regarding the used nodes, the utilization of the Palladio resource containers was throughout the whole simulation at 100 %. With a longer experiment duration, it decreases but soon is zero. We think that is a bug because Palladio ran out of memory and stopped to simulate sending data. Under these circumstances, it is no surprise that the different scaling thresholds do not show any difference regarding the response time.

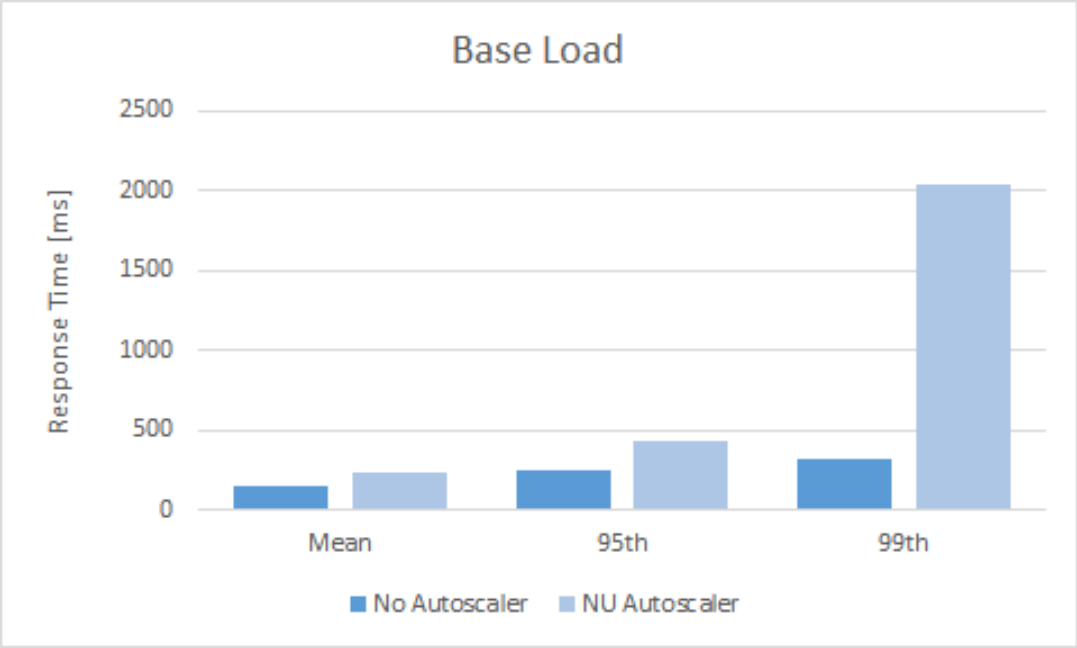

Figure 5.29: Comparison QVTo Scaling Policy implementation with no scaling through the baseload.

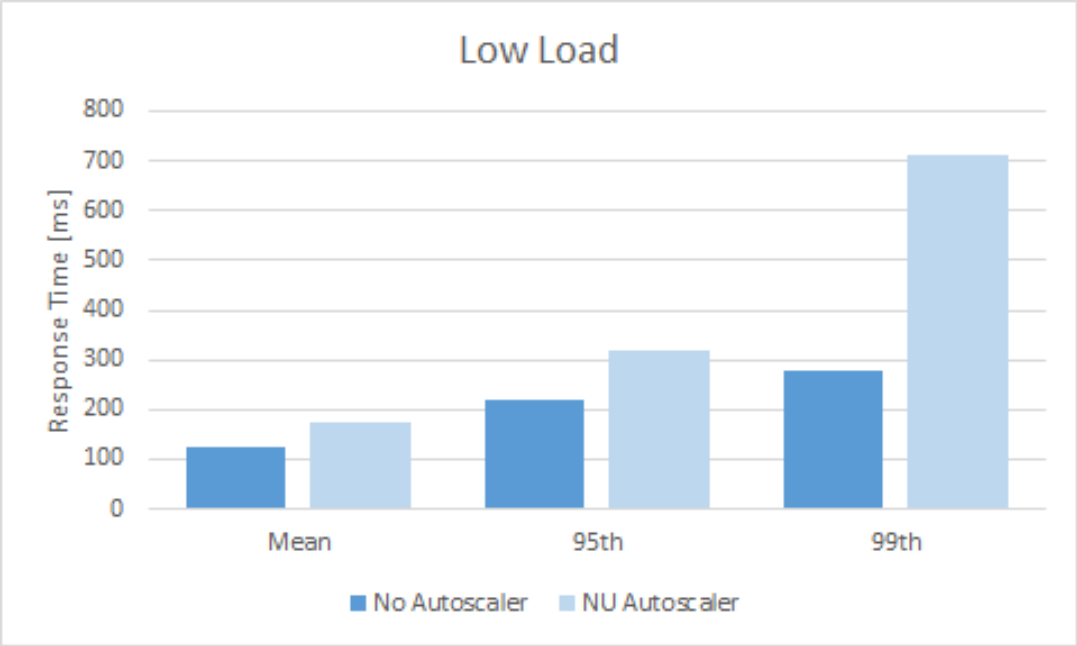

Figure 5.30: Comparison QVTo Scaling Policy implementation with no scaling through the low load.

Now we proceed from the autoscaler implementation to the modeling without autoscaling. The results of the three different load scenarios do not differ much, but there is still a significant difference visible. The prediction of the medium load has the best response times and the prediction of the baseload the worst.

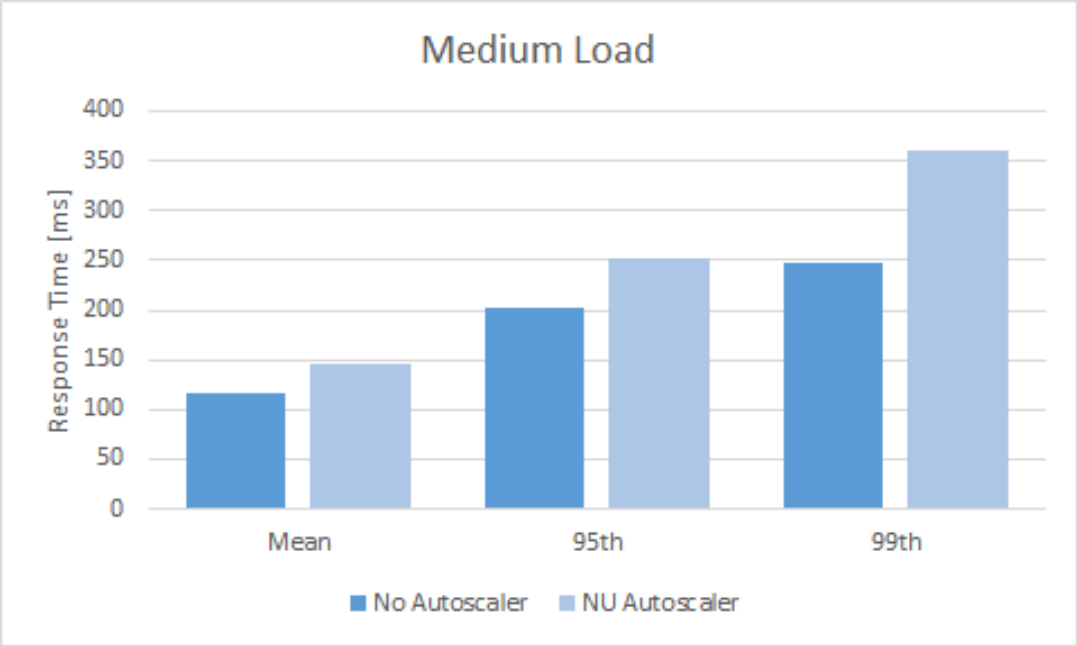

Figure 5.31: Comparison QVTo Scaling Policy implementation with no scaling through the medium load.

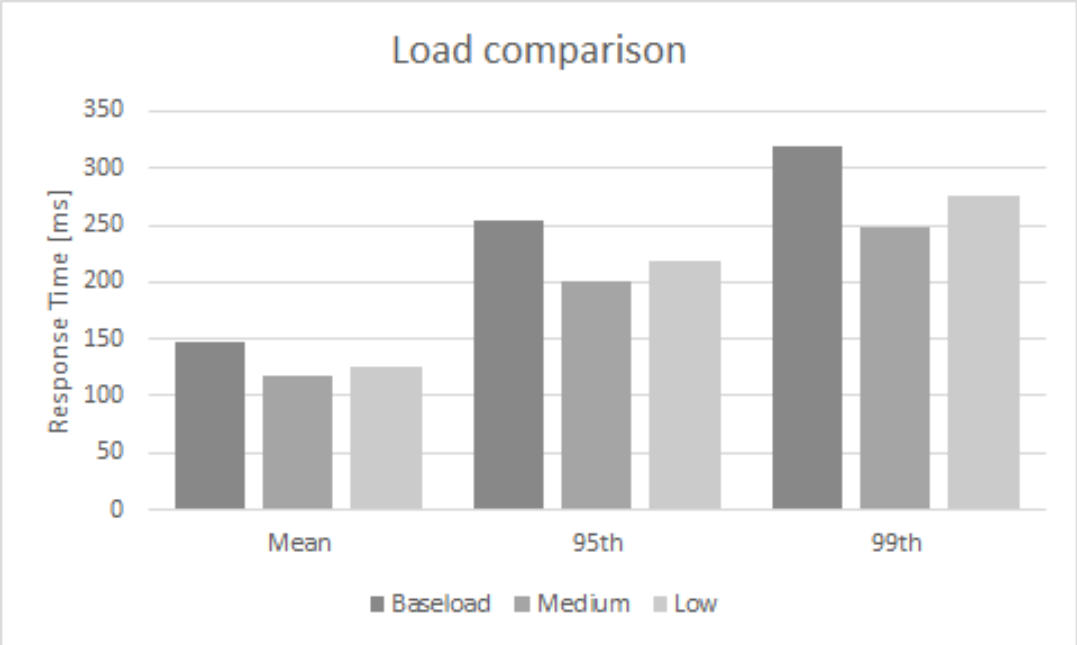

Figure 5.32: Comparison simulation results of the three load scenarios.

6 Evaluation

In this chapter, we evaluate the result with an analysis of the research questions. Then we discuss, based on the results, our hypothesis regarding acceptance or rejection. In addition, we discuss what threatens the validity of our results.

6.1 Analysis

Our thesis has two viewpoints, feasibility, and explainability. We first analyze the feasibility part and, after that, the explainability.

6.1.1 RQ1- Feasibility

One goal of this thesis is to demonstrate feasibility of Palladio simulating scaling policies through accurate simulations. That was the focus of the following sub-questions.

RQ1.1

RQ1.1 asks how accurate are the simulation results.

In short, they are not accurate. At least for our experiment scenario and consecutively this thesis.

We simulated single parts of the experiments without the transient phases because of several limitations and shortcomings. We failed to model the complete experiment scenario because we could not create a working Usage Evolution. We can compare the numbers of single load stages between the transient phases. Here we calibrated the inter-arrival time of the sending operations with a fitted distribution of the experiment data.

The Gatling data to compare the simulations is the data where we fitted the distribution. That means we excluded the ramp-up but did not remove the zeros and the outliers. The results of this approach show an acceptable accuracy.

If we consider the effect of our fitting procedure, the results for predicting the low and medium load stage are acceptable. Figure 6.1 and Figure 6.2 show that the mean response time was missed by a range of 50 to 100 ms. The exponential distribution has more problems with the 95th and the 99th percentile, but still, it is not a multiple of the response time.

The accuracy performance changes for the baseload prediction. Here the simulation predicts a response time twice as high. We have to consider that the exponential fitting fitted less well than for the other cases, compare Figure 5.28.

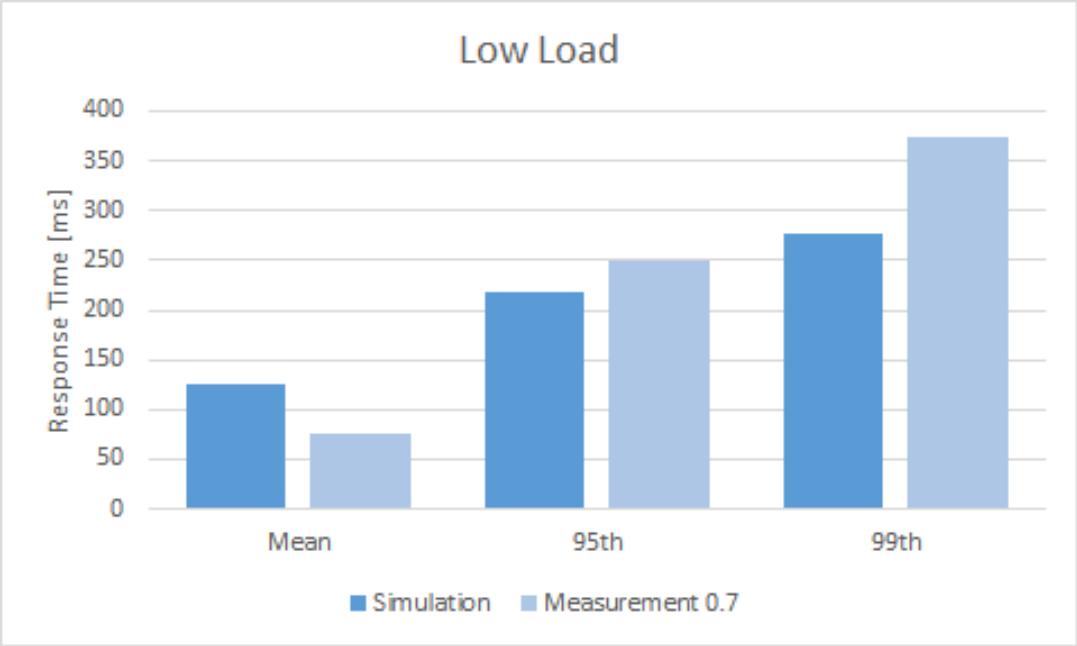

Figure 6.1: Comparison of the simulation with the measurement - low load.

All things considered, we can say that with the Distribution fitting, we can create reasonably accurate performance predictions. At least if the distributions fits. As a result we think the prediction shown in figure 5.32 reflects how good the likelihood of the fitting is. That means creating a good fitting is a bigger concern than the performance of Palladio.

Still, it is no scaling policy model which would allow us to weigh up design decisions. We can not asses which scaling policy configuration or style produces the better tail latencies.

RQ1.2

RQ1.2 asks if the best performing scaling policy configuration is predicted correctly.

As described in RQ1.1, we cannot predict the scaling policies correctly, so we failed to predict the best-performing policy.

Regarding the trend correctness, which means to pick the best performing policies, our results show doubts that the distribution covers that correctly. For instance, the prediction result says the medium load has the best response times, which is the opposite in reality. However, this is not an exclusive problem of the scaling policy simulation. As said before, we assume this is related to the exponential distribution, as the histogram of the baseload is not fit well.

At least our comparison of the autoscaler simulation with the normal implementations validates our implementation to a certain degree. And it shows the potential of Palladio to predict the transient phase during the scaling process, compare Figure 5.29 ff. In addition, the result confirms the findings of Klinaku et al. [KHB21]. It is possible to model scaling policies with Palladio, but the accuracy remains unclear.

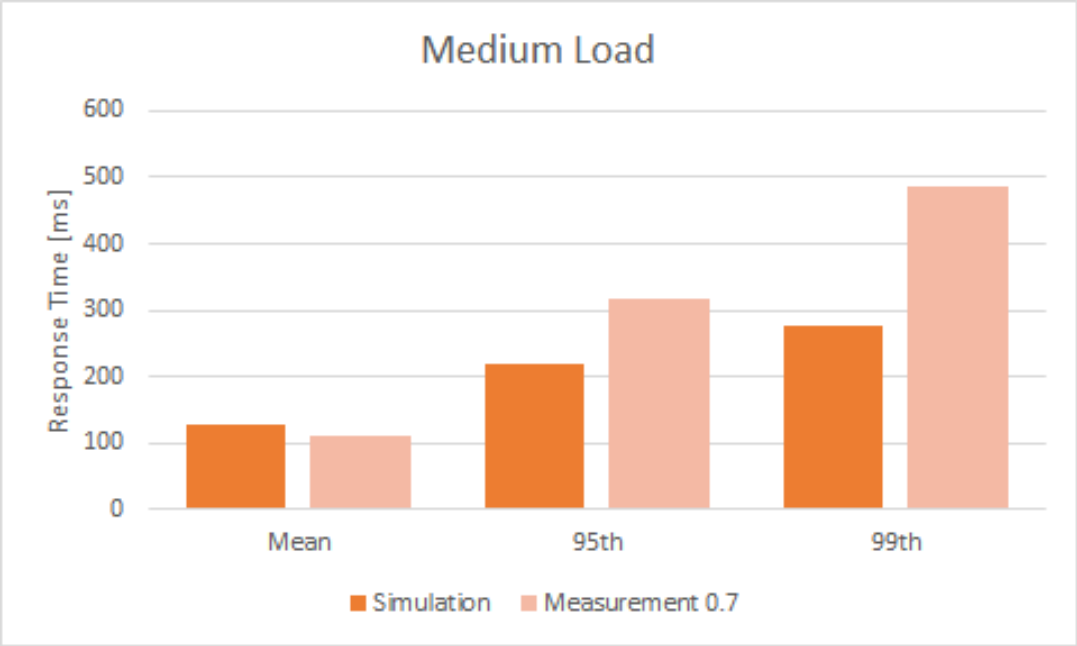

Figure 6.2: Comparison of the simulation with the measurement - medium load.

RQ1.3

RQ1.3 asks if the best performing scaling policy style is predicted correctly.

We cannot answer this research question because we had to skip this experiment due to time constraints caused by the problems we faced.

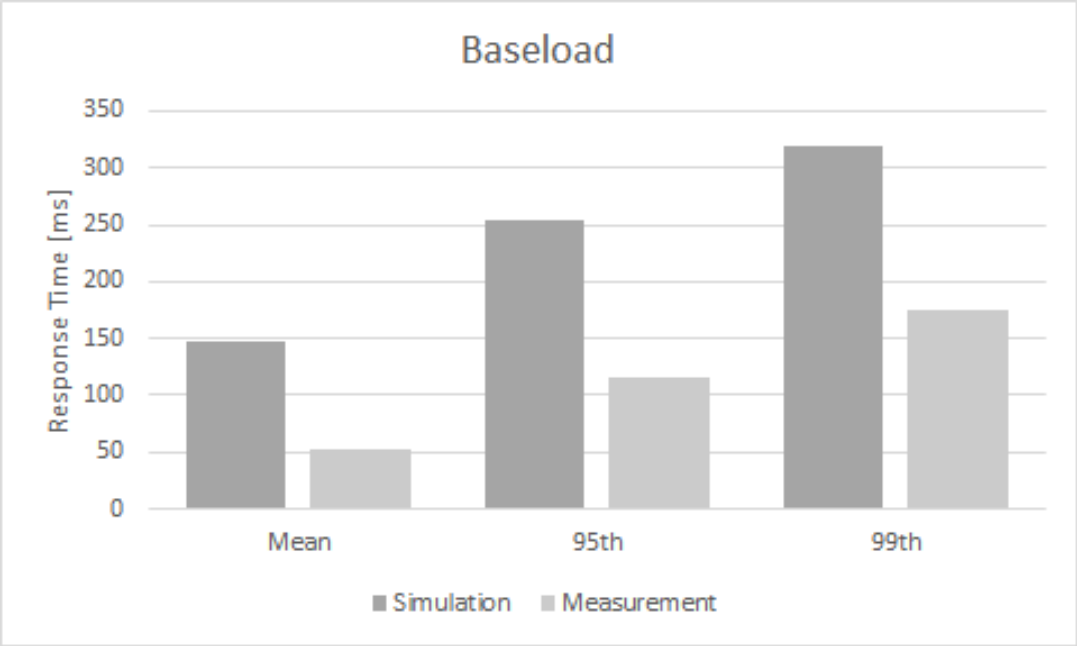

Figure 6.3: Comparison of the simulation with the measurement - baseload.

6.1.2 RQ2- Explainability

RQ2 asks if Palladio can aid in improving the explainability of scaling policies. We can confirm that by showing Palladios potential to retrace scaling decisions. Both Figure 6.4 and Figure 6.5 show how Palladio can aid the explainability of self-adaption decisions.

At the bottom of both graphics, the activity of an additional resource is shown by the red dots. In Figure 6.4, we can see that the response time decreases significantly when the additional resource is active. The improved response time is the effect of the self-adaption triggered by the utilization of the system.

The point in time when self-adaption is triggered is shown in Figure 6.5. The first time the red dot in the upper chart is above the utilization of 0.8 (scaling threshold), the additional replica becomes active. In addition, the trend of increasing utilization is stopped, and the metric is below the scale-in threshold. The two minutes quiescence period prevents scale-in action until roughly 450.

That shows, Palladio helps to effectively trace back the reason and the effect of a self-adaption. At least in this scenario and under the premise, the viewer is able to interpret the graphs correctly. Therefore we can answer RQ2 with yes, it does theoretically aid the explainability of scaling policies.

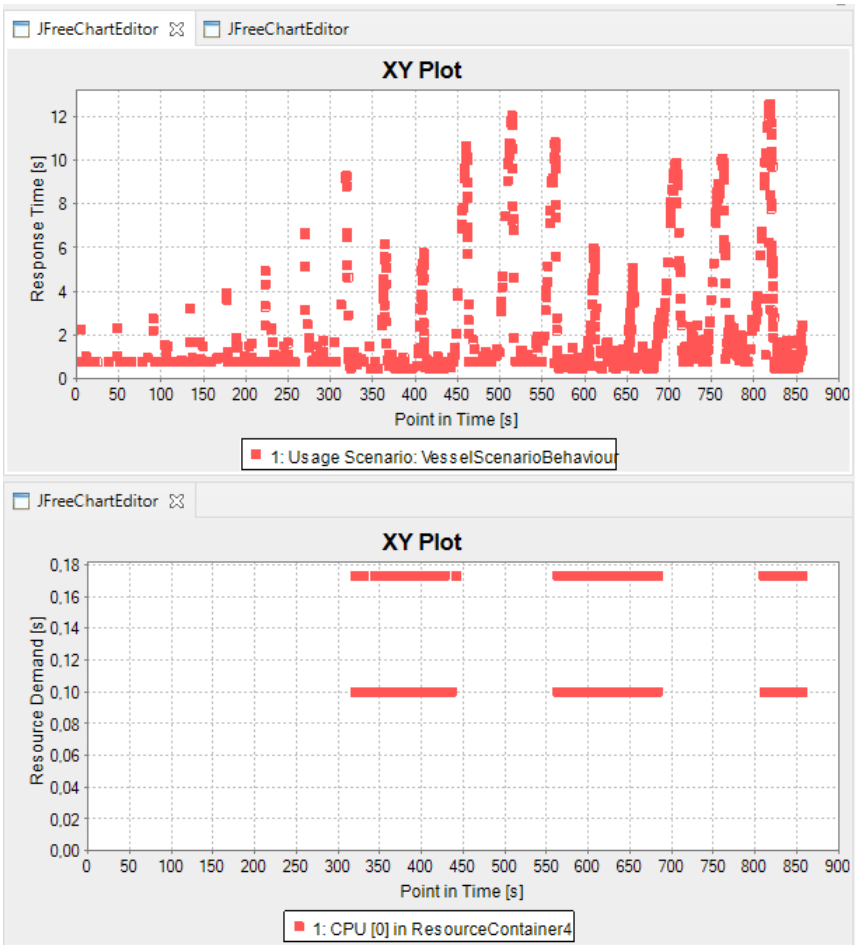

Figure 6.4: The overall response time together with the activity of an additional replica.

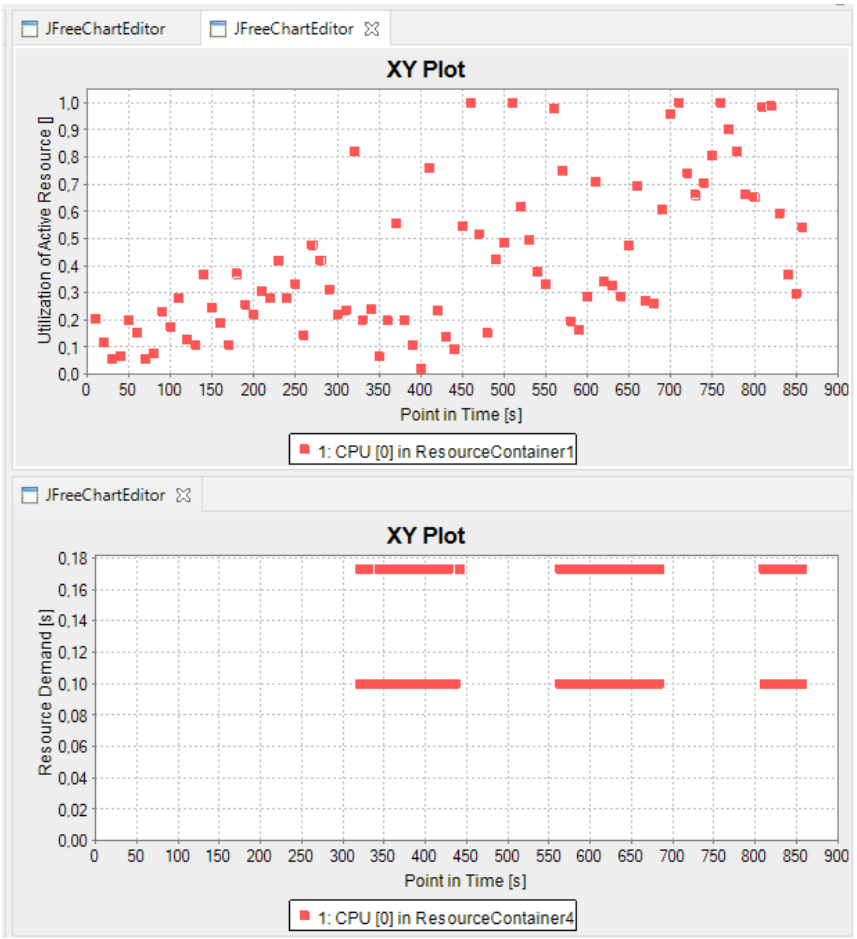

Figure 6.5: The utilization together with the activity of an additional replica.

6.2 Discussion

After seeing the result, we have to reject our hypotheses for RQ1. For our experiment setting, it was not feasible to simulate scaling policies to the full extent.

Even more, we have to note that we were not even able to model the NUA and the elasticity experiment. That is worse than having an inaccurate performance prediction and questions the effectiveness of Palladio in that regard. It makes not much sense to compare it to the MAPE of Aslanpour et al. [ATTG21]. However, we have to consider several validity threats that prevent the generalization of our findings. There are good reasons to believe Palladio can model scaling policies, as others showed to some extent. Even if it failed for our experiment.

At least five factors influenced the outcome of our thesis and could be the reasons for the failed simulation.

The first is the effect of ProtoCom, which could be the reason why we obtained a permanent utilization of 100 % of the Palladio resource container. However, this is contradicted by the results of the individual load phases. Those response time results are not off by a multiple. However, these results also do not seem to show a system that is utilized to 100 %.

That leads to the second factor, which is potential model flaws. The model was created by the MoSaIC project and was constantly reviewed. Some flaws were addressed, e.g., the database becoming the bottleneck because the resource demand was set too high. But there is no silver bullet to validate the model other than running experiments as ours. At least if there is enough time to do that.

The third factor is related to that but does go beyond model flaws. The MoSaIC system is a messaging system using Kubernetes. A Palladio model that includes both concepts was not evaluated before. And now we added a third concept, which does not make it easier to isolate problems.

The fourth factor is Gatling and its load distribution for the MoSaIC experiment scenarios. That was the reason why we could not use the Usage Evolution that increases the number of users. Or it was at least the reason that caused problems to proceed. Probably with more time and further experiments, there is a fix either the load distribution of Gatling or to evolve the inter-arrival time.

The last and fifth factor is Palladios memory usage during the simulation execution. First, it is not ideal that parts of the simulation can run short without interrupting the simulation. That creates the risk to obtain misleading results. It is not better when Palladio crashes because the Java Runtime Environment does not have enough memory available to continue. We observed that phenomenon on two machines with at least 16 GB RAM available. At this stage, it is not clear how to interpret that. Thats probably prevents an extensive analysis of systems with a low inter-arrival time or requires more powerful hardware.

An investigation and solving some of the factors could eliminate the problems we faced and confirm the feasibility of modeling scaling policies with Palladio.

In contrast, we can accept our hypothesis that Palladio helps to aid the explainability of scaling policies. The validity threat is that we miss proof that our findings are generalizable and hold after a controlled experiment.

7 Conclusion

The goal of this thesis was to evaluate the architecture-based simulations of Palladio. More specifically, to evaluate Palladios elasticity capabilities through modeling scaling policies.

We will summarize our work in this chapter, outline the benefits, limitations, lessons learned, and suggest future work.

7.1 Summary

This thesis saw through the selection, implementation, benchmarking, modeling, and simulation of scaling policies. To assess the feasibility of creating simulations that help software developers to evaluate design decisions.

One of the key aspects of work is the complexity of selecting a suited scaling policy. Scaling policies come in different styles and configurations. Therefore stakeholders have a wide range of possibilities to choose a policy. Fuzzy requirements and different states of knowledge delay the process further and are additional reasons why a common knowledge base would help to ease that process. That knowledge could be enriched by effective simulators that outline the advantages and drawbacks of each solution.

The implementation of a simple scaling policy is easier, be it the HPA or something like the Node Utilization Scaler. In most cases, the difficult part is gluing things together and testing the correctness of the implementation. That is also a possible area of application for simulations.

Benchmarking a cloud environment is easy to execute with a load test tool as Gatling and with a structured approach like doubling the user parameter stepwise. Such a scalability experiment as the one we conducted lightens the design of an elasticity experiment.

The elasticity experiment and the evaluation of two different scaling policy configurations worked but cost a lot of time. In the end, we could assess that the lower utilization threshold is the more effective solution regarding the response time.

The hardest part of our thesis was the modeling part. Palladio is a powerful tool that offers a ton of possibilities. But along the possibilities, the complexity rises to infinity. It was not problematic to create a scaling policy but hard to get it running. And still, details prevented us from simulating the elasticity experiment scenario.

Therefore we have to reject that it is feasible to model scaling policies with Palladio. Parts of the simulator showed its potential, and compared to the effort it took to benchmark and run the elasticity experiment, the simulation finished in no time. However, that applies only under the condition that the model exists and works.

The explainability is related to that because it might be easier to observe statistics from Prometheus on the real Kubernetes cluster. Nevertheless, implementing and debugging the QVTo autoscaler implementation worked with the monitor capabilities of Palladio.

7.2 Benefits

The biggest benefit and contribution is the documented experience with Palladio. Hopefully, everyone who intends to work with Palladio to create scaling policies can rely on our experiments.

We wrote a detailed profile guide and published it on GitHub. That should help to create QVTo scaling policies in the future. We also proposed how the HPA formula could be implemented. This implementation could be realized if the modeling problems are solved. In addition, [KHB21] already mentioned the problems the scaling policy definition with QVTo has. In our case, we think that it is even worse regarding the effort. However, the biggest problem is not QVTo itself. Of course, it is a shortcoming that you cannot store values straight forward, but the crucial thing is the fact that the workaround is hardly if at all documented. This is something that was mentioned by Klinaku et al. [KBB19], even though they said that in a different context. So that is a problem in general for PCM, Palladio, and QVTo. In addition, we documented the shortcomings that prevented us from assessing the accuracy of the scaling policy model. That contribution should also help in future work.

Further, we contributed a scalability and elasticity experiment for the MoSaIC system. The first will help everyone who will work with the MoSaIC system in the future because we benchmarked the system. With that knowledge, it is clear which amount of vessels the system can handle and that the ramp-up should be excluded or chosen wisely. Additionally, our elasticity experiment showed that a lower scaling threshold leads to a better response time latency. And we documented that the utilization metric is problematic because it did not reflect the high response times during our scalability experiment. We also helped to detect flaws in the implementation. The MoSaIC project can use that to design better scaling policies in the future.

We also helped create a more useful Gatling load test implementation for experiments. The concept with a baseload and an additional load in a second ramp-up makes it easier to conduct experiments. Before, the implementation allowed only incrementing using the same step size, which is cumbersome. That is the same for the experiment duration, which was a constant before. [KHB21] already mentioned the problems the scaling policy definition with QVTo has, in our case we think that it is even worse regarding the effort. However, the biggest problem is not QVTo itself. Of course it is a shortcoming that you cannot store values straight forward, but the crucial thing is the fact that the workaround is hardly if at all documented. This is something that was mentioned by [KBB19], even though they said that in a different context. So that is a problem in general for PCM, Palladio and QVTo.

The key benefit is, that we described the process to that in detail and hopefully future work is not confronted by the same hurdles as we were. So that in future it will be easier to use QVTo to specify more complex scaling policies. See the guide in the appendix.

Further, we contributed a scalability and elasticity experiment for the MoSaIC system. The first will help everyone who will work with the MoSaIC system in the future, because we benchmarked the system and it is now clear which amount of vessels the system can handle and that the ramp-up should be excluded.

We adapted the Gatling load test, so that it is more useful for experiments. The concept with a base load and an additional load in a second ramp-up makes it easier to conduct experiments. Before, the implementation allowed only an increase of the same step size, which is cumbersome. That is the same for the experiment duration, which was a constant before.

Moreover, we showed that it works to calibrate the inter-arrival rate with a distribution.

7.3 Limitations

One limitation for our elasticity experiment is that we excluded the setOnline from our simulation because we had problems simulating the whole scenario. We addressed the distortion risk of the result by excluding the ramp-up from the experiments. That is the phase where setOnline was executed. Although doing that does not erase the influence of this method. In addition, we had to rush the selection of the distribution because of time reasons. Better distributions than gamma or exponential may exist because we did not evaluate them in much depth or search for other possibilities.

Another limitation is, we monitored one resource container/node in the scaling policy threshold. It is probably possible to aggregate several measuring points into one measuring specification. But we found no documentation for that. Aggregating means that the autoscaler monitors an average among all active nodes or pods. We were not able to model the behavior like that. That is also related to the problem of why we could not model the HPA.

Further on that issue, the HPA and our NUA also observed the utilization metric of more than one node/pod or Palladio resource container. The load is distributed equally in the SEFF. Therefore, we expect an equal utilization distribution on the available nodes. However, we cannot be sure about that. To sum it up, most limitations touch the model and Palladio part.

7.4 Lessons Learned

A lesson learned is that we showed that it is possible to model more complex scaling policies with a threshold, constraints, and more than one additional resource instance. Further, it is no problem to realize the different scaling policy styles from Klinaku et al. [KHB21] with Palladio. For the proactive scaler, it is not trivial, but as we modeled the HPA mechanism, we showed that it is possible to model a more proactive scaling policy.

We noticed during the elasticity experiment that the utilization metric is relatively weak to reflect the response time. For instance, while the system seemed overstressed, the monitored utilization was still below the scaling threshold. Therefore the autoscaler did not scale, and the high response time did not improve.

We saw that the MoSaIC system, or more precisely the setOnline method had problems with timeouts. That resulted in failures, and the devices never sent data. With no recovery method, e.g., a control loop that tries setOnline, this is not ideal and probably unrealistic. The problem arose during the second ramp-up when we tried a longer ramp-up time than 20 seconds and got worse with every second of ramp-up time more. We suspect that this is because with a five sec ramp-up time setOnline was executed during a period where the system was nearly idle. Hence, it was not a problem to avoid the time out, but with a better-distributed load, the system is not idle anymore. Therefore it is more likely that setOnline times out. This behavior is not severe for our purpose. But it is worth to be taken into consideration for further experiments. In addition, the get campaigns method for the expert user timed out too. Therefore, we decided to leave the expert users out of our experiment scenario. Not just because of that issue but also to avoid unnecessary co-founding factors. However, we reported the behavior, and it became clear that the number of campaigns is the issue. That resulted in a fix by restricting the number of campaigns.

Another lesson learned is the behavior of Gatling. We were surprised about the oscillation in the sendData during our experiments. It seems that with more sendData invocations the oscillation got bigger, while the failed setOnline executions amplified the oscillation. At least there was a correlation because as soon the setOnline failed, sendData executions of these devices were missing. At this time window, Gatling did react to that by redistributing the sendData of the remaining devices. As a result, the created unbalance stayed and affected the system, which correlated to a peak load visible in the response time. Adding to that, we also saw why Aslanpour et al. [ATTG21] opted for traces instead of creating their load. Creating your load is beneficial because it gives you full control. But it is also way more effort to get to a reasonable load.

The next lesson learned affects the Usage evolution. We saw that the Usage Evolution for the inter-arrival time is either not working properly or not used correctly. We found no documentation on that matter. Therefore it is hard to tell what exactly went wrong. Although it seems likely that the inter-arrival time with the capability to define StoEx (Stochastic Expressions) is not suited for the Usage Evolution. The population field of the closed workload, which specifies the Usage Evolution evolves there, only allows numbers and does not offer the possibility to enter StoEx. In addition, we learned that specifying a gamma distribution as inter-arrival time did not work as well as with an exponential distribution.

Regarding Palladio, we learned that low interarrival times seem to challenge Palladio respectively hardware. Palladio either crashed because there is insufficient memory for the Java Runtime Environment to continue. Another possibility would be that Palladio stopped the data collection of the response time early, for the same reason. Another Palladio problem was that the newer versions of Palladio (5.0 and nightly) do not allow the usage of the utilization metric, which resulted in a NullPointerException.

Overall, after conducting various experiments on a cloud system, we can confirm the need for working and accurate performance predictions. The process of executing experiments takes a lot of effort. It would save a lot of that when the simulator could answer at least some questions faster.

7.5 Future Work

We suggest that future work should investigate the following issues.

It could be that more powerful hardware will solve in the memory problem we described in the previous chapters. It seems unrealistic to expect a more efficient implementation soon. An alternative is to assess possible workarounds to avoid the problem. We ran out of time to try that. That applies also to the remaining four factors we described in our discussion that prevented us from modeling our scaling policy for the MoSaIC system.

We assume that future work is also needed to successfully model the HPA. In addition, we propose an investigation of possibilities to preserve a mapping from the Kubernetes pods to the resource containers. It is important to show that because otherwise, it is not possible to determine whether Palladio is ready to model Kubernetes applications that are using the HPA. And if it does not work, to adapt or extend Palladio to allow metrics in nested systems. For example, to monitor an allocation context too.

There are intentions to extend SimuLizar by the Slingshot approach [KB20], among other things the plan is to allow analyses between layers. At this point, it would also make sense to address the shortcomings the current solution has, apart from the documentation of existing elements.

One of them is the Usage Evolution. It would be interesting to know what the Usage Evolution does when it should evolve an open workload. Was that considered during the development, and should that be possible? In addition, we think it is interesting to investigate if it is possible to evolve the inter-arrival time specified through StoEx, also if it is possible with distributions.

Along with that, the problems with gamma distributions should be further investigated. First how to use it correctly, and secondly if the results with such a distribution are significantly better than with the exponential distribution.

Another item for future work is explainability. We initially planned to do more work in that area to support our perception that Palladio simulations allow retracing adaption decisions. That was skipped because of time reasons. So that has to be investigated further. One possibility to do that would be a survey with stakeholders that work with scaling policies to furnish evidence on that matter.

Bibliography

[ATTG21] M. S. Aslanpour, A. N. Toosi, J. Taheri, R. Gaire. "AutoScaleSim: A simulation toolkit for auto-scaling Web applications in clouds". In: *Simul. Model. Pract. Theory* 108 (2021), p. 102245. DOI: 10.1016/j.simpat.2020.102245. URL: https://doi.org/10.1016/j.simpat.2020.102245 (cit. on pp. 16, 23–25, 30, 32, 44, 78, 82).

[BBH18] B. Burns, J. Beda, K. Hightower. *Kubernetes*. Dpunkt, 2018 (cit. on pp. 5–9).

[BBL17] S. Becker, G. Brataas, S. Lehrig. *Engineering Scalable, Elastic, and Cost-Efficient Cloud Computing Applications: The CloudScale Method*. Springer, 2017 (cit. on p. 19).

[BBM13] M. Becker, S. Becker, J. Meyer. "SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems". In: *Software Engineering 2013*. Ed. by S. Kowalewski, B. Rumpe. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 71–84 (cit. on pp. 16, 20, 21, 26, 30).

[BDH08] S. Becker, T. Dencker, J. Happe. "Model-Driven Generation of Performance Prototypes". In: *Proceedings of the SPEC International Workshop on Performance Evaluation: Metrics, Models and Benchmarks*. SIPEW '08. Darmstadt, Germany: Springer-Verlag, 2008, pp. 79–98. ISBN: 9783540698135. DOI: 10.1007/978-3-540-69814-2_7. URL: https://doi.org/10.1007/978-3-540-69814-2_7 (cit. on p. 11).

[Bec17] M. W. Becker. "Engineering self-adaptive systems with simulation-based performance prediction". PhD thesis. Universität Paderborn, 2017 (cit. on pp. 20, 26).

[BSM20] D. Balla, C. Simon, M. Maliosz. "Adaptive scaling of Kubernetes pods". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–5. DOI: 10.1109/NOMS47738.2020.9110428 (cit. on p. 5).

[CRB+11] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya. "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms". In: *Softw. Pract. Exper.* 41.1 (Jan. 2011), pp. 23–50. ISSN: 0038-0644. DOI: 10.1002/spe.995. URL: https://doi.org/10.1002/spe.995 (cit. on p. 16).

[DD15] M. L. Delignette-Muller, C. Dutang. "fitdistrplus: An R Package for Fitting Distributions". In: *Journal of Statistical Software* 64.4 (2015), pp. 1–34. DOI: 10.18637/jss.v064.i04. URL: https://www.jstatsoft.org/index.php/jss/article/view/v064i04 (cit. on p. 65).

[GBMP13] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013). Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond, pp. 1645–1660. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.01.010. URL: https://www.sciencedirect.com/science/article/pii/S0167739X13000241 (cit. on p. 1).

[GLV19] J. Greenyer, M. Lochau, T. Vogel. *Explainable Software for Cyber-Physical Systems (ES4CPS): Report from the GI Dagstuhl Seminar 19023, January 06-11 2019, Schloss Dagstuhl*. 2019. arXiv: 1904.11851 [cs.SE] (cit. on pp. 21, 30).

[GSFP20] F. Ghirardini, A. Samir, I. Fronza, C. Pahl. "Model-Driven Simulation for Performance Engineering of Kubernetes-style Cloud Cluster Architectures". In: Dec. 2020. DOI: 10.1007/978-3-030-63161-1 (cit. on p. 16).

[HGGG12] R. Han, L. Guo, M. M. Ghanem, Y. Guo. "Lightweight Resource Scaling for Cloud Applications". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 2012, pp. 644–651. DOI: 10.1109/CCGrid.2012.52 (cit. on p. 5).

[HKR13] N. R. Herbst, S. Kounev, R. Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not". In: *10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 23–27. ISBN: 978-1-931971-02-7. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst (cit. on p. 3).

[HPA] *HPA Horizontal Pod Autoscaler - Documentation*. Kubernetes. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (cit. on pp. 9, 10).

[KB20] F. Klinaku, S. Becker. "The Slingshot Approach: Model-Driven Engineering the Coordination of Autoscaling Mechanisms for Elastic Cloud Applications". In: Dec. 2020, pp. 158–165. ISBN: 978-3-030-63160-4. DOI: 10.1007/978-3-030-63161-1_13 (cit. on p. 83).

[KBB19] F. Klinaku, D. Bilgery, S. Becker. "The Applicability of Palladio for Assessing the Quality of Cloud-Based Microservice Architectures". In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. ECSA '19. Paris, France: Association for Computing Machinery, 2019, pp. 34–37. ISBN: 9781450371421. DOI: 10.1145/3344948.3344961. URL: https://doi.org/10.1145/3344948.3344961 (cit. on pp. 1, 26, 27, 29, 80).

[KHB21] F. Klinaku, A. Hakamian, S. Becker. "Architecture-based Evaluation of Scaling Policies for Cloud Applications". In: 2021 (cit. on pp. 4, 5, 26–29, 35, 37, 72, 80, 81).

[KHK14] J. v. Kistowski, N. R. Herbst, S. Kounev. "Modeling Variations in Load Intensity over Time". In: *Proceedings of the Third International Workshop on Large Scale Testing*. LT '14. Dublin, Ireland: Association for Computing Machinery, 2014, pp. 1–4. ISBN: 9781450327626. DOI: 10.1145/2577036.2577037. URL: https://doi.org/10.1145/2577036.2577037 (cit. on p. 19).

[KK16] S. Kim, H. Kim. “A new metric of absolute percentage error for intermittent demand forecasts”. In: *International Journal of Forecasting* 32.3 (2016), pp. 669–679. ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2015.12.003. URL: https://www.sciencedirect.com/science/article/pii/S0169207016000121 (cit. on p. 32).

[Kou17] S. Kounev, J. O. Kephart, A. Milenkoski, X. Zhu, eds. *Self-Aware Computing Systems*. Berlin Heidelberg, Germany, 2017 (cit. on p. 21).

[LB14] S. Lehrig, M. Becker. “Approaching the Cloud: Using Palladio for Scalability, Elasticity, and Efficiency Analyses”. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days, SoSP 2014, Stuttgart, Germany, November 26-28, 2014*. Ed. by S. Becker, W. Hasselbring, A. van Hoorn, S. Kounev, R. H. Reussner. Vol. 2014/05. Technical Report Computer Science. University of Stuttgart, 2014, pp. 141–151. URL: http://www.performance-symposium.org/fileadmin/user%5C_upload/palladio-conference/2014/papers/paper9.pdf (cit. on pp. 19, 26).

[Leh14] S. Lehrig. “Applying Architectural Templates for Design-Time Scalability and Elasticity Analyses of SaaS Applications”. In: *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*. HotTopiCS ’14. Dublin, Ireland: Association for Computing Machinery, 2014. ISBN: 9781450330596. DOI: 10.1145/2649563.2649573. URL: https://doi.org/10.1145/2649563.2649573 (cit. on p. 1).

[LZ11] S. Lehrig, T. Zolynski. “Performance Prototyping with ProtoCom in a Virtualised Environment: A Case Study”. In: Nov. 2011 (cit. on p. 11).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011 (cit. on p. 3).

[Mur04] R. Murch. *Autonomic Computing*. IBM Press, 2004. ISBN: 013144025X (cit. on p. 20).

[NMM+18] F. Nieuwenhuizen, D. Mikalkinas, L. Merz, D. Schütz, B.-A. Vu, B. Pushpanathan, M. Tepeli. “Tooling: Improved Management for Monitor Repositories and Measuring Points in Palladio”. In: (Nov. 2018) (cit. on p. 58).

[NYK+20] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, S. Kim. “Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration”. In: *Sensors* 20.16 (2020). ISSN: 1424-8220. DOI: 10.3390/s20164621. URL: https://www.mdpi.com/1424-8220/20/16/4621 (cit. on pp. 5, 9, 10).

[RBB+11] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, M. Kuperberg. *The Palladio Component Model*. Tech. rep. 14. Karlsruher Institut für Technologie (KIT), 2011. 193 pp. DOI: 10.5445/IR/1000022503 (cit. on pp. 17–19).

[RBH+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, K. Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016. ISBN: 026203476X (cit. on pp. 1, 16).

[SK16] C. Stier, A. Koziolek. “Considering Transient Effects of Self-Adaptations in Model-Driven Performance Analyses”. In: *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. QoSA’16. Venice, Italy: ACM, 2016. DOI: 10.1109/QoSA.2016.14 (cit. on pp. 21, 26, 29).

[SMC+08] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, S. Babu. “Cutting Corners: Workbench Automation for Server Benchmarking”. In: *USENIX 2008 Annual Technical Conference*. ATC’08. Boston, Massachusetts: USENIX Association, 2008, pp. 241–254 (cit. on p. 31).

[TM18] A. Taivalsaari, T. Mikkonen. “On the development of IoT systems”. In: Apr. 2018, pp. 13–19. DOI: 10.1109/FMEC.2018.8364039 (cit. on p. 1).

[Tur18] J. Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018 (cit. on p. 10).

All links were last followed on December 10, 2021.

A Appendix

Distribution code

Listing A.1 R code

```
# Precondition: remove zeros with excel excel filter before importing
# Otherwise the fitting will fail
elastData <- read.csv(file = '../ElastData.csv')
library("fitdistrplus")

# remove outliers
outliers <- boxplot(elastData$V5)$out
ElastData <- elastData[-which(elastData$V5 %in% outliers),]

# convert to seconds
gamma <- fitdist(ElastData/1000, "gamma")
exp <- fitdist(ElastData/1000, "exp")

# plot to check the fitting
plot.legend <- c("gamma", "exponentional")
denscomp(list(gamma,exp), legendtext = plot.legend)

# output for the Usage Model Calibration
summary(gamma)
summary(exp)
```

Experiment results

Vessels	2000	4000	6000	8000	10000
Mean	283	288	367	426	352
95th	660	652	975	1165	820
90th	548	556	764	931	683
SD	186	186	287	335	233

Table A.1: Scalability experiment. Response time in ms of send data with baseload 2000 on four nodes. Values after the floating point were removed.

Devices	500	1000	1500	2000	2500
Mean	281	221	236	259	438
95th	624	563	493	504	962
90th	555	452	423	442	888
SD	177	156	133	133	294

Table A.2: Scalability experiment. Response time of send data with base load 500 on one node. Values after the floating point removed.

Load	Low		Medium	
Threshold	0.7	0.9	0.7	0.9
Mean	332	578	181	186
95th	862	1677	424	425
99th	2751	3624	580	564
SD	396	658	144	126
Nodes used	3	2	2	2

Table A.3: Measured data from the elasticity experiment. Mean, 95th, 99th and SD in ms.

Load	Base		Low		Medium	
Case	Simulation	Measurement	Simulation	Measurement	Simulation	Measurement
Mean	146	52	127	77	117	109
95th	254	116	219	250	201	317
99th	318	176	276	374	248	485
SD	56	132	51	82	46	201
Nodes used	4	2	4	2	4	3

Table A.4: Measured and simulated data from the elasticity experiment. Mean, 95th, 99th and SD in ms.

Profile Guide

This tutorial explains the use of an EMF/MDSD Profile in a PCM project.

Use

Using the ScalingPolicyProfile works as follows.

Right click on *ScalingPolicyCatalogue.architecturaltemplates* and run with *run configurations*.

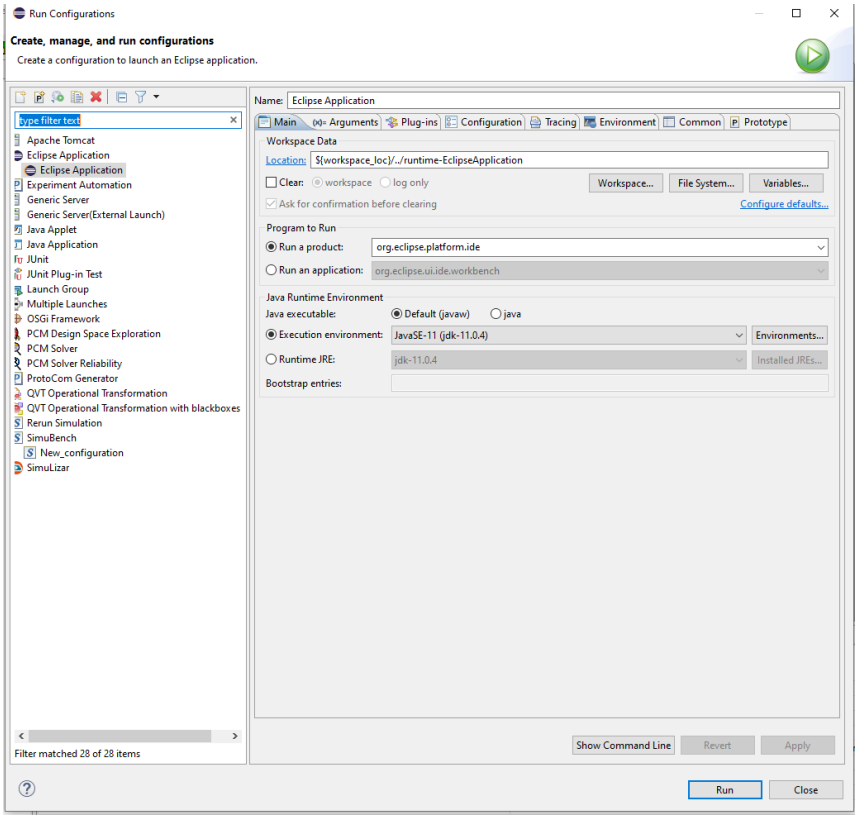

Figure A.1

Then Eclipse Application and create a new one. Change Program to Run / Run a product to *org.eclipse.platform.ide*.

After that you can work with the Profile.

The easiest way to apply is:

1. Open the system model
2. Right click on it
3. MDSD Profiles
4. First select Apply/Unapply Profiles -> ScalingPolicyProfile -> add

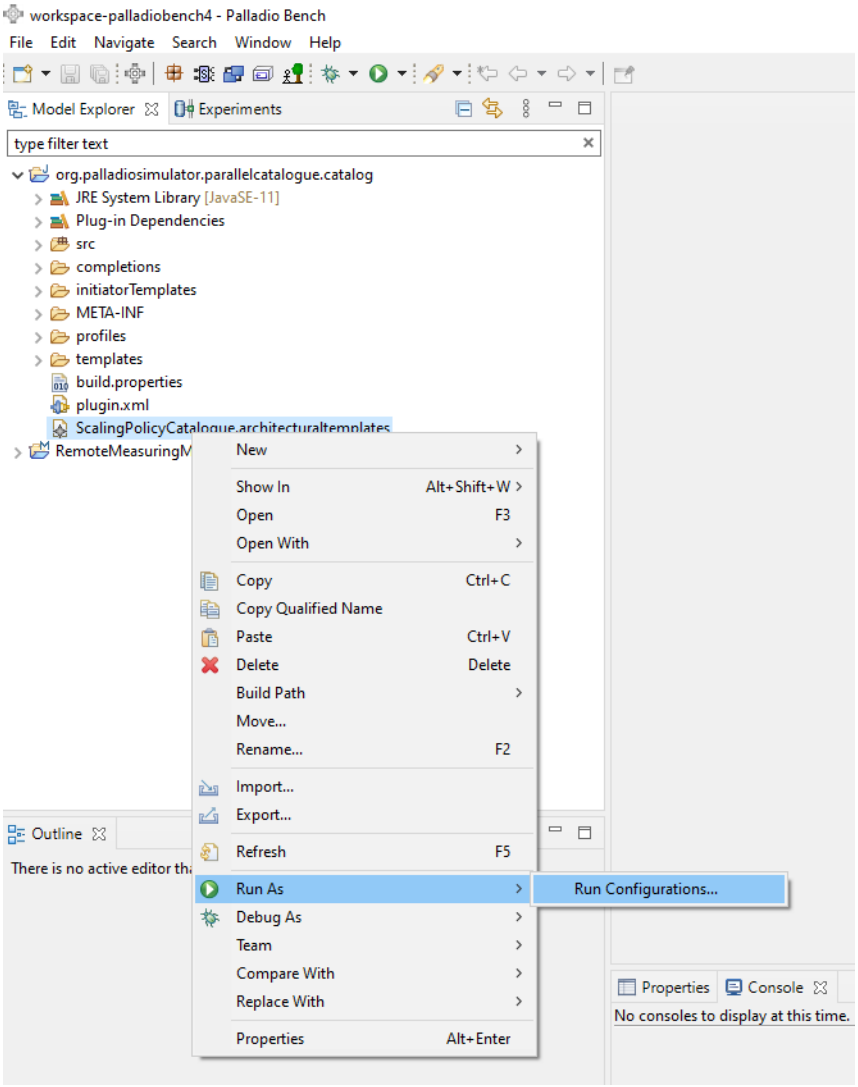

Figure A.2

Listing A.2

```
[pool-4-thread-167] WARN : Rule application failed with message: The system has no
ScalingPolicyProfile applied!
```

5. Second select Apply/Unapply the Stereotype -> ScalingPolicySystem -> add
6. Refresh the project with F5

It should look like this.

Be aware of the fact that the profile application is unstable. It often does not work correctly. For instance, if the profile is still applied after opening the workspace, it will not work. You need to open it without one applied. If not, the script will fail at the check hasProfileApplied.

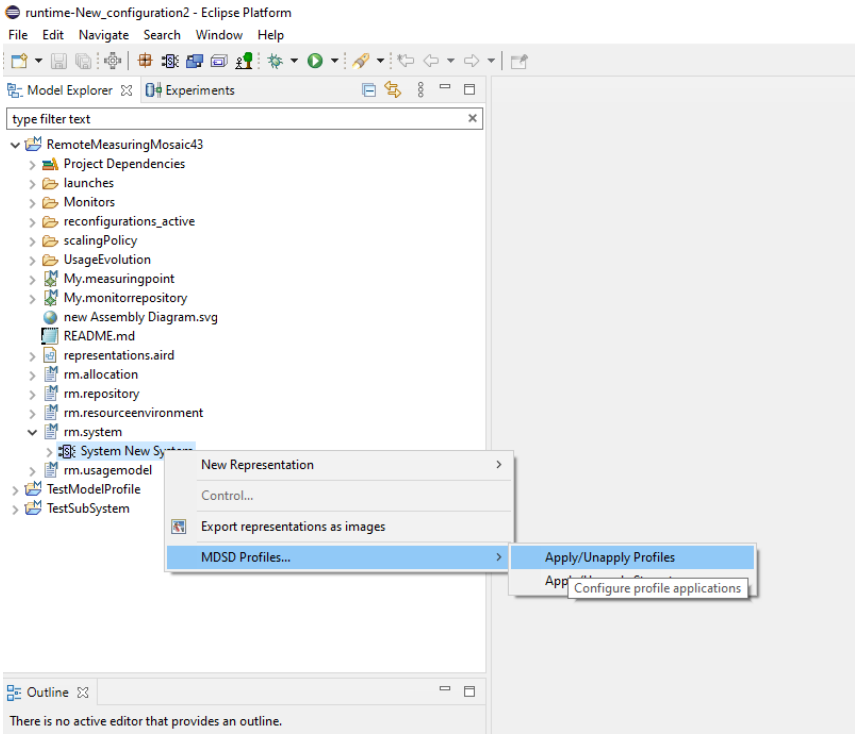

Figure A.3

Trouble Shooting

If an error is thrown during the execution of the QVTo self-adaption, then a restart of eclipse is needed. Often when *getTaggedValue* is printed out as invalid - that is an indication to try a restart. Also, we were not able to use Integer fields respectively to use *setTaggedValue* (there was a warning that the call is ambiguous), which might be fixed in future versions.

So it is important to debug. Also, if Palladio version conflicts appear, in our case, a model created with the nightly version did not work in 4.3, even after replacing the *representation.aird*.

To test the profile, see *TestModelProfile* and *NodeUtilization* scaling policy. By executing the QVTo script with SimuLizar, you can test and debug.

Developing

For developing we refer to the sdq wiki. Read the MDSD [1], EMF [2] and AT [3] guides. The latter describes how to create an EMF profile in the first part and in the second the AT. Also follow the AT guide instructions to create the *.architecturaltemplates*.

Be careful that all IDs are correct, watch the console of the outer Eclipse.

[1] https://sdqweb.ipd.kit.edu/wiki/MDSDProfiles
[2] https://sdqweb.ipd.kit.edu/wiki/EMF_Profile_Definition
[3] https://sdqweb.ipd.kit.edu/wiki/Architectural_Templates

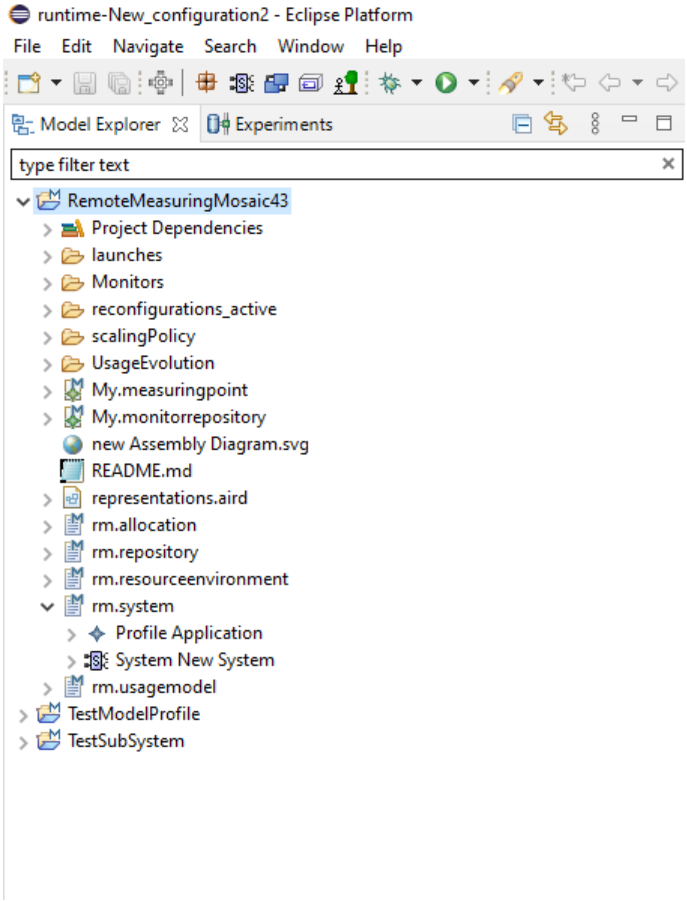

Figure A.4

The Profile is publicly available on [4].

[4] https://github.com/PhilippGruber/ScalingPolicyProfile

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.

---

place, date, signature