Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis

Automated Performance Regression Detection in Microservice Architectures

Nils Wenzler

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Dr.-Ing. André van Hoorn Teerat Pitakrat Cor-Paul Bezemer
Commenced:	March 23, 2017
Completed:	September 23, 2017
CR-Classification:	C.2.4, D.2.8

Abstract

The emergence of microserive architectures has lead to a need for software performance techniques which cater to the needs of these architectures. Microservice achrictectures are architectures which are build out of small independent processes which communicate by use of language independent interfaces such as REST. Microservice environments challenge common software performance engineering techniques by their highly distributed nature, their rapidly changing systems and their extensive use of virtualization and containers. Detecting performance changes of a system during development, so-called performance regression detection, is a valuable addition to the development and maintenance of software systems. How software performance regression detection can be performed in microservice architectures is the non-trivial question, on which this thesis focuses to answer.

To reach this goal, in a first step extensive research on by the microservice orchestration technology Kubernetes and the containerization technology Docker provided software performance metrics is performed. Results show that some major performance metrics can not be considered to be stable concerning redeployments of the microservice system. They suggest that performance metrics of a microservice instance can be highly impaired by other microservices running on the same node.

A second step empirically evaluates the performance of existing performance regression detection techniques in the context of a microservice environment. After a thoroughly comparison of the different approaches, selected approaches were implemented and their performance was evaluated empirically in the test setup. The results show that some approaches are not applicable or show a bad performance in the microservice setup. Although none of the approaches performed well enough for practical application, two of the approaches showed promising results, which could lead to enabling performance regression detection in microservice archtectures.

Kurzfassung

Das Aufkommen von Microservice Architekturen hat zu einem Bedarf an neuen Software-Performanz-Techniken, die an die Eigenschaften dieser Architekturen angepast sind, geführt. Microservice Architekturen sind Architekturen die aus kleinen und unabhängigen Prozessen aufgebaut sind, die über sprachenunabhängige Schnittstellen wie REST kommunizieren. Microservice-Umgebungen stellen auf Grund ihrer massiven Verteiltheit, den sich schnell und häufig ändernden Systemen und ihrer Verwendung von Virtualisierung und Containerisierung eine Herausforderung für Software-Performanz-Ingeniuere dar. Performanceänderungen eines Systems zu erkennen, so genannte Performance-Regressions-Erkennung, ist eine wertvolle Ergänzung zur Entwicklung und Wartung von Softwaresystemen. Die nicht-triviale Frage, wie Performance-Regressions-Erkennung im Umfeld von Microservice-Umgebungen durchgefürht werden kann, steht deshalb im Fokus dieser Arbeit.

Um Antworten auf diese Frage zu finden, werden zunächst die von der Microservice-Orchestatrationstechnologie Kubernetes und der Containerisierungstechnologie Docker zu Verfügung gestellten Performanz-Metriken untersucht. Die Ergebnisse dieser Arbeit zeigen, dass manche Microservice-Performanz-Metriken im Bezug auf Redeployments des selben Systems auf dem selben Cluster nicht als stabil angesehen werden können. Sie suggerieren, dass Performanz-Metriken einer einzelnen Microservice-Instanz massiv durch andere Microservice-Instanzen auf dem selben Knoten beeinflusst werden können.

In einem zweiten Schritt wird die Performanz existierender Performanz-Regressions-Erkennungsverfahren im Microservice-Umfeld empirisch untersucht. Nach einem gründlichen Vergleich der unterschiedlichen Verfahren, wird eine Auswahl der Verfahren nachimplementiert und ihre Performanz in einem Test-Microservice-System mit künstlichen Regressionen evaluiert. Die Ergbnisse zeigen, dass manche Verfahren nicht anwendbar sind oder sehr schlechte Ergebnisse erzielen. Obwohl keins der untersuchten Verfahren für eine praktische Anwendung gut genug wäre, zeigen zwei der untersuchten Verfahren vielversprechende Ergebnisse, die zielführend für die Entwicklung von Performanz-Regressions-Erkennung im Microservice Architekturen sein könnten.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1. Motivation

Microservice architectures promise to reduce complexity, to give the possibility of scaling independently, to remove and deploy independent parts of the system easily, to support usage of different frameworks and languages, to increase the overall system elasticity and finally to improve the resilience of the system [Ama+15]. Many companies already use this by service oriented architectures inspired architectural style. To software performance engineers these architectures pose a lot of new challenges and a need for software performance applications which cater to the specific needs of microservice environments has emerged [Hei+17].

Performance regression detection is one of those software performance applications. Performance regression detection detects significant changes in the performance of a software system during development and helps to avoid performance decreases. It basically answers the question whether the overall systems performance has changed because of the most recent changes to the code base.

The approach of tackling software performance during development has per se some advantages compared to dealing with performance issues at the end of the development of a project. On the one hand, it can be very challenging to try to target performance issues at the end of software development because the sources of the bad performance may be hidden somewhere in the system. Oppositely, if the developer is notified directly when a performance decrease, a so-called performance regression was introduced during development, it should be less challenging to find and resolve the issue. On the other hand, stands the fact that nowadays software systems, and microservice systems in special, are subject to continuous change caused by techniques such as continuous delivery. It therefore sounds reasonable to monitor the performance of a software system on a base of the different changes during development.

This setting leads to the central question of this thesis's work: How and how well can performance regression detection be realized in the context of microservice environments?

Some of the challenges in the context of microservice systems which have to be beat to enable performance regression detection in such an environment are dealing with the distributed nature of the systems independent microservices, understanding the performance behavior of the microservice systems, researching the available performance metrics of microservice environments and their properties, finding software performance techniques for testing elasitc systems, and deciding on how to incorporate the different metrics of the different microservice instances into the performance regression detection. Because of the distributed nature of microservice systems, virtualization and containerization are commonly used technologies. These themselves add challenging aspects to software performance engineering tasks. Basic foundations to software performance engineering, such as the performance metrics behave differently in containerized applications. E.g., this thesis shows that performance metrics of containers are impaired by other microservices running on the same physical node.

Although this thesis is not able to answer all of the questions concerning performance regression detection in microservice achitectures, it offers in-depth research and first results to some of those challenging questions. In this work, a reference microservice system is used for an empirical study on the possibilities of performance regression detection in microservice environments. Extensive experimentation on the different available performance metrics of microservices is performed and their properties are investigated. The stability of measurement results during test runs and in comparison between different deployments of the same system are researched. In this work, it is shown that most of the measurements were not of a normal distribution, that the performance metrics did not show any unexpected deviations during single test runs, but that massive deviations of up to 16% of performance measurement results were observed in the comparison of different deployments of the same microservice system and on the same cluster. The reason for these deviations was found in scheduling decisions of the used microservice orchestration technology Kubernetes.

To further investigate the possibilities of performance regression detection in microservice architectures, in-depth research on existing performance regression detection approaches was performed. After a thoroughly comparison of the different approaches, selected approaches were implemented and their performance was evaluated empirically in the test setup. The results show that some approaches are not applicable or show a bad performance in the microservice setup. None of the existing approaches performed on a level which would be usable in a productive environment. Two of the implemented approaches show promising results and may lead to solutions for performance regression

detection in the context of microservices. The best approach could detect 67% of the injected regressions with an accuracy of 73%.

Finally, this work shows possible future directions for research to enable performance regression detection in microservice environments.

1.2. Thesis structure

The remainder of this thesis is structured as follows:

Chapter 2 – Foundations explains the basic concepts needed to understand this work. It offers a short introduction into the theory of load testing, regression detection and microservice architectures.

Chapter 3 – Related work presents a detailed description of the existing research work on the different performance regression detection approaches.

Chapter 4 – Comparison and implementation of approaches offers a reasoning concerning which existing approaches were chosen to be evaluated in a microservice environment. It offers a collection of criteria by which the selection was performed and compares the different approaches concerning these criteria.

Chapter 5 – Evaluation shows the setup and methodology of the evaluation of this thesis. Furthermore, the findings of the experiments concerning microservice metrics behavior and performance of the performance regression detection approaches in a microservice environment are described and afterwards discussed.

Chapter 6 – Threats to validity lists possible threats to the validity of the findings presented in this work. For the different threats, a short evaluation and steps to minimize the risks are presented.

Chapter 7 – Conclusion gives a short walk through the thesis, its evaluation and its findings. An outlook concerning possible future research is given.

Chapter 8 – Acknowledgements lists the people without whom this work would not have been possible or who gave valuable inputs during the creation of this thesis.

The original measurements of this work were published at [Wen17b]. The original code base of this work was published at [Wen17a]. This code base does only offer the exact implementation of the prototype. It neither offers further documentation nor a performance regression detection tool which could be used in a productive environment.

Chapter 2

Foundations

This chapter explains the basic concepts needed to understand the following work. It offers a short introduction into the theory of load testing, regression detection and microservice architectures.

2.1. Performance testing

Performance testing is a type of software testing, which strives to determine the responsiveness, throughput, reliability, and/or scalability of a software system under a given load [Mei+07]. A load is a, most commonly artificially, produced set of operations which are performed on the system under test (SUT) to observe its behavior when working. Performance testing helps to identify bottlenecks in a system, supports performance tuning and enables evaluating compliance of a system with service level agreements (SLA). SLAs specify requirements to the product build in a software development project. Examples for such requirements could be guaranteed up time, certain response time limits for a specified number of users or a guaranteed throughput of documents per hour.

In general, there are four different types of performance testing: load tests, stress tests, performance tests and capacity tests [Mei+07]. Those types of testing mainly differ in the goal that they strive to reach. Performance tests target to determine the speed, scalability or stability of the SUT. Scalability describes the system's ability to adapt to increasing workloads by usage of additional resources (see Section 2.7.1). Load tests strive to evaluate the SUTs behavior under normal and peak conditions. Stress tests evaluate the behavior of the SUT under loads which exceed those of normal or peak conditions. Capacity tests target the question of how many users/transactions per time slot the SUT is able to support while still meeting its performance requirements.

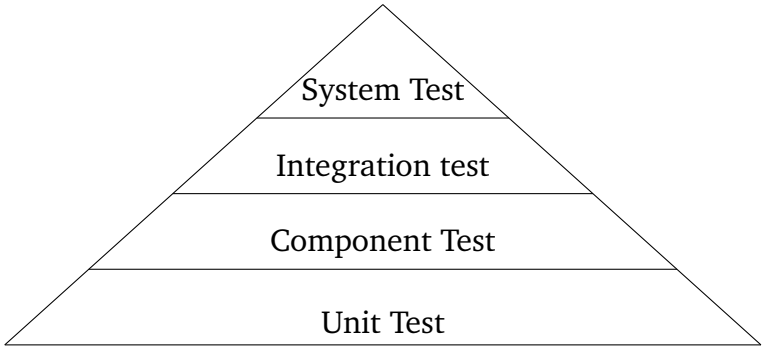

Figure 2.1.: The different levels of load testing. Adaption of Mike Cohn's test pyramid [CG09].

Independent of the type of a performance test, there are different levels at which load testing can be performed. Figure 2.1 shows how those levels build on each other. While unit tests, focus on testing small code passages and classes, component tests focus on testing the behavior of whole components. Integration tests focus on testing the interaction between single components while system tests strive to test the overall systems performance. The main focus of this thesis will be on testing on a system and component level.

2.2. Performance regressions

Functional regression tests are a commonly used method for assuring that a newer version of a software system still fulfills the functionality of the older version [LL13]. Such tests are called (functional) regression tests. Analogically, performance regressions describe a significant change of performance compared to an older version of the software system. Although functional regression tests are well established, performance regression tests are not as commonly performed in practice.

Examples for typical performance regressions are [Ngu+12] [Sha+15]:

Increasing memory usage Adding a (large) field in a very often used object will lead to a prominent increase of overall memory usage.

Increasing CPU usage Additional calculations or algorithms with a bad run-time will increase the CPU load of the SUT.

Increasing I/O access times Since storage device accesses are in comparison to memory accesses more time consuming, introducing an increased amount of such (blocking) I/O accesses will decrease the performance of the SUT.

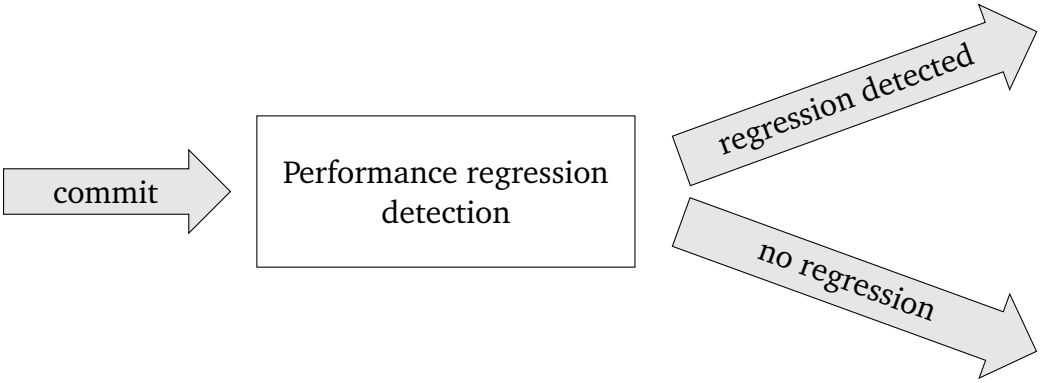

Figure 2.2.: The process of performance regression detection

Increasing network usage In comparison to memory accesses, network I/O is more time-consuming and can have an overall negative impact on inter-microservice communication. Introducing an increasing amount of network calls will decrease the performance of the SUT.

Unnecessary system prints System prints, which are sometimes used for debugging, may slow down parts of the system, since the output depends on slow I/O operations.

Wrong configurations A wrong configuration of the system components may slow down the system significantly. Examples of such configuration errors may be: number of available threads in a thread pool, number of parallel connections to a database or resource limits for single processes.

2.3. Performance regression detection

Performance regression detection deals with the detection of performance regressions. Although approaches for performance regression tests exist, they are uncommon compared to functional regression tests. Figure 2.2 shows the general process of performance regression detection. After a new commit, a change in the software system is submitted, the performance regression evaluates the new system's performance and inspects whether a performance regression was added or not. Performance regression detection compares the performance measurements of a new version v_i of a software system to the performance measurements of a subset of the earlier observed versions $v_0, ...v_{i-2}, v_{i-1}$ of the software systems. In between different versions the code base, the functionality of the software and even the load of the evaluation may change.

2.4. Evaluation of performance regression detection approaches

For later evaluation of the performance of the performance regression detection approaches the following metrics are introduced. There are four general results which a single evaluation of a performance regression detection approach can have. The approach can report a performance regression when the system indeed has a performance regression. This is called a true positive (TP). The approach may report that there is no performance regression when the system's performance indeed has not changed. This is called a true negative (TN). For a perfect approach which does not make mistakes, these two results would be sufficient, but there are two cases in which the approach could make errors. If it reports no regression although in reality one could observe a regression, this kind of error is called a false negative (FN). Finally, if the system reports a regression although there is no performance anomaly, we observe a false positive (FP) [DG06].

Out of those four kinds of results, three different metrics are commonly calculated. The precision, the recall and the F-measure.

The precision describes how many of the reported regressions are indeed regressions. It fits to the common understanding of the word precision.

$$Precision = \frac{TP}{TP + FP}$$

The recall describes how many of the real regressions were reported to be a regression.

$$Recall = \frac{TP}{TP + FN}$$

Additionally there is the F-measurement which combines both metrics.

$$F = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

For all three metrics, the results range from 0 to 1, with one being the optimal case.

2.5. Anomaly detection

Performance regression detection is a special use case of anomaly detection. Anomaly detection is the problem of finding patterns, so called anomalies, in data. Anomalies are

patterns which are considered to be not normal. In the context of this work, performance regressions can be considered to be anomalies concerning software performance.

In a survey, Chandola, Banerjee, and Kumar [CBK09] give an overview over the different kinds of anomaly detection and offer basic algorithms for the different types. If not otherwise stated, the information of this section refers to [CBK09].

Defining what is normal behavior, how to deal with an adapting definition of normal behavior and how to deal with noise in the input data, are challenges in anomaly detection. Noisy data is data which is corrupted or distorted.

Different techniques deal with different kinds of data. Some techniques may only work for data in form of attributes (Mining performance regression testing repositories serves as an example), while others use ordinal or continuous data (Student t-test based performance regression detection serves as an example).

The following section explains what general types of anomalies exist, how anomaly detection techniques are trained to recognize normal and uncommon behavior and finally which general techniques are used to detect anomalies.

In general, there are three categories of anomalies:

Point anomalies are anomalies where a single measurement can be considered to be an anomaly in reference to the remaining data set.

Contextual anomalies are anomalies, where the single measurement itself can not be considered to be an anomaly since the value of it can be common. Contextual anomalies are deviating concerning their context. While a temperature of 30 degree Celsius can be considered normal in the context of summer, a temperature of 30 degree Celsius in winter would be considered to be a contextual anomaly.

Collective anomalies are anomalies concerning related data samples, which can only be considered uncommon because of their existence as a collection. A possible example for such anomalies would be tracking actions in the field of intrusion detection and seeing a suspicious series of events in the tracking logs.

The performance regression detection approaches, which are described in this thesis vary in the kind of anomalies they detect. Although the classification is to some amount subject of interpretation, the approach which is described in Section 3.2.8 could be seen as focusing on contextual anomalies while the approach of Section 3.2.5 could be seen as focusing on point anomalies. No approach presented in this work deals with collective anomalies.

Anomaly detection approaches need to build a model of normal behavior. The process of building such a model is called training. According to Chandola, Banerjee, and Kumar [CBK09] three different kinds of techniques concerning training exist: supervised,

semi-supervised and unsupervised training. Supervised training offers training data with labels for normal as well as for anomalous data. Semi-supervised training offers training data which only shows normal behavior. Finally, unsupervised techniques do not use labeled data at all. They most commonly assume frequent patterns to be normal. The performance regression detection approaches in this thesis are setup in an environment where they may use old performance measurements, which are considered to be of normal behavior. Therefore, performance regression detection, as understood in this work, can be categorized into the section of semi-supervised training.

Another classification is given by the general approach which the different techniques use:

- classification based
- nearest neighbor based
- clustering based
- statistical techniques
- information theory
- spectral theory

The research concerning the different existing approaches in the field of software performance regression detection are hard to categorize. Nonetheless, some approaches can be considered to be in the category of the statistical techniques.

2.6. Microservice architectures

Many modern software businesses focus on the use of microservice architectures. Being a fine grained Service Oriented Architecture (SOA), they promise to cater to the needs of cloud computing and continuous delivery (CD) [BHJ16]. A microservice architecture is built out of single independent processes, so called microservices. The main reasons for the emergence of microservice architectures are their promise to reduce complexity, to give the possibility of scaling independently, to remove and deploy independent parts of the system easily, to support usage of different frameworks and languages, to increase the overall system elasticity and finally to improve the resilience of the system [Ama+15]. Scalability is a prerequisite for elasticity and describes the degree to which a system is able to sustain increasing workloads by making use of additional resources [LEB15]. Elasticity adds the aspects of how fast how often and at what granularity the system adapts. Resilience describes “ the ability of a system to sustain external and

internal disruptions without discontinuity of performing the system's function or if the function is disconnected, to fully recover the function rapidly" [HBRM16].

Container virtualization is commonly used in microservice architectures. It allows to run applications like microservices on one single system side-by-side in isolated containers. Containers offer virtualization on the level of the operation system and are therefore more lightweight, compared to full server virtualization [Ama+15]. They promise to be easy to setup, since all dependencies of a software are defined and bundled into the container.

Since microservice architectures are mostly used together with CD approaches, microservice environments are rapidly changing and it is hard or even impossible to find a steady state for performance regression detection [Hei+17]. Therefore, approaches for performance regression detection in microservice architectures should target that fact in a certain way. The overall lack of software performance engineering approaches for microservice systems [Hei+17] is one of the main motivations for this thesis.

The remainder of this section will give a short introduction to the terminology of the used container orchestration tool Kubernetes [Kubd].

Kubernetes is an open-source system which offers deployment and management functionalities for a microservice system. In this thesis, Kubernetes is used for deploying the microservices. This short subsection tries to explain the terminology and most basic concepts of Kubernetes, so that future references to such terminology will be understandable.

A single microservice is in most cases build out of a containerized application, which allows independent deployment and execution. Since in some cases a microservice might be built out of several containers, Kubernetes base unit is a pod. A pod represents one single instance of a microservice in the system.

Since pods may be redeployed or several instances of one pod may run simultaneous to make up for high demand of this microservice, there is a need to have one central point to ask for instances of one kind of microservice. In Kubernetes, this concept is called a service. Commonly requests are not issued to a pod but to a service which forwards the request to one of the available microservice instances associated with one service.

To control the number of instances of a single microservice, so called replica sets allow to horizontally scale it. Horizontal scaling describes the process of providing more or less instances of a microservice to adapt to a changing load. Opposed to that, vertical scaling describes the process of making more or less resources available to one single existing instance of a microservice.

In terms of hardware, there are so called nodes. A node corresponds to one single system in the cluster. Such nodes may be independent hardware systems or may be a set of virtual machines.

When mentioning a redeployment in this work, it means to delete all pods, services and replica sets of the system under test and to reinstantiate them afterwards.

2.7. Performance metrics

According to the IEEE standard 610.12, metrics are a quantitative measure, to which degree a system, component or process possesses a given attribute. Metrics are used to evaluate an attribute of a system and to help comparing attributes between different systems. Performance metrics are metrics which measure aspects which help when evaluating performance of a system.

Typically, there are four main types of system performance metrics: CPU utilization, memory utilization, network I/O and disk I/O [Ngu+12]. Additional measurements of metrics are collected on application level. A performance counter is a concrete dataset of a given performance metric. A typical load test may collect thousands of performance counters [Sha+15]. Therefore, tooling support is needed when software performance analysts try assessing the performance of a system.

The performance attributes scalability, elasticity and resilience are of special importance for evaluating microservices. Scalability is a performance attribute, which describes the degree to which a system is able to sustain increasing workloads by making use of additional resources [LEB15]. Elasticity adds the possibility of decreasing available resources and is based on how much time the system needs to perform such an adaption [LEB15]. Resilience describes “ the ability of a system to sustain external and internal disruptions without discontinuity of performing the system’s function or if the function is disconnected, to fully recover the function rapidly” [HBRM16].

Since those performance attributes can not be directly measured, software performance engineers use special metrics and combinations of metrics to evaluate scalability, elasticity and resilience.

2.7.1. Measuring scalability

Scalability is a performance attribute, which describes the degree to which a system is able to sustain increasing workloads by making use of additional resources [LEB15]. In contrast to scalability, it contains the possibility of reducing available resources. Tsai,

Huang, and Shao [THS11] propose to observe performance change and performance variability for evaluating scalability in a cloud computing environment. This thesis assumes that microservice environments and cloud computing environments are comparable in their requirements to scalability. Tsai et al. build their scalability metric upon the following definitions.

- The waiting time T_w is defined as the sum of the queuing time T_q and the execution time T_e.

  $$T_w = T_q + T_e$$

- C_R is the sum over all resources of the product of all resource allocations of a resource R_i and the time T_i the resource is used.

  $$C_R = \sum_i R_i * T_i$$

- The performance/resource ratio PRR is the product of the inverse waiting time T_w and the inverse C_R value.

  $$PRR = \frac{1}{T_w} * \frac{1}{C_R}$$

- Finally the performance change PC is defined as

  $$PC = \frac{PRR(t)W(t)}{PRR(t')W(t')}$$

  with $PRR(t)$ as the performance resource ratio of the system under a certain work load $W(t)$ and $PRR(t')$ and $W(t')$ being the performance resource ratio and the work load at a different time t'. In a scalable system, PC should have a value close to 1. A value of 1 means, that the resource need per work instance does not change with working loads.

- The performance variance PV is the standard deviation of the performance change for multiple test runs with the same constant workload.

  $$PV = E[(PC_i - \frac{1}{n}\sum_{i=1}^{n} PC_i])^2]$$

  A scalable system should have a performance variance close to zero. A value of zero represents that the performance change is constant.

2.7.2. Measuring elasticity

Scalability is a prerequisite for elasticity [HKR13]. While scalability describes the ability of a software system or component to adapt to increasing workloads by making usage of additional resources, elasticity adds the aspects of how fast, how often and at what granularity the system or component adapts. Herbst, Kounev, and Reussner [HKR13] propose a metric for elasticity which uses the concept of underprovisioned and overprovisioned states. Underprovisioned states are states in which the system or component does not have enough resources available. Overprovisioned states are states in which the system has more resources available than needed. To recognize such states, Herbst et al. propose to build a matching function which gives information on when the system should scale down or up. Afterwards they use metrics like average time for the system to leave an underprovisioned state and the average amount of underprovisioned resources in the system to evaluate the overall elasticity. Another approach for measuring elasticity, is given in the work of Islam et al. [Isl+12]. It is the most commonly referenced approach [LEB15]. They as well consider under- and overprovisioned states, but use a cost penalty for evaluating the elasticity. In general Islam et al. integrate over cost resulting out of the difference between resources which were needed and resources that were available. They propose different approaches to evaluate a cost of under- and overprovisioning. The approach for underprovisioning uses quality of service metrics to calculate a violation metric regarding the SLAs. For overprovisioning they use the actual cost, which the additional resources cost.

2.7.3. Measuring resilience

While all the three software attributes, scalability, elasticity and resilience, are challenging to measure, resilience seems to be the toughest one to measure. The most common definition [Pas+15] of resilience is, the one given by Hollnagel, who defines it as: “the intrinsic ability of a system to adjust its functioning prior to, during, or following changes and disturbances, so that it can sustain required operations under both expected and unexpected conditions” [Hol13]. Pasquini et al. [Pas+15] mention in their work, that although this definition is good for understanding the general concept of resilience, parts of the resilience engineering community argue that it is not well suited for measuring resilience. Strigini [Str12] states in his work that resilience and dependability are broad concepts, covering several attributes and therefore several possible metrics are available. One possible metric which targets the dependability is the overall system availability in a realistic setup. Such a metric is very dependent on the SUT and its load, it therefore does not perform well when comparing different systems. Furthermore, the results when artificial load is being used as a reference highly depend on what kind of disturbances of

the system are present in the load. In addition, the possibility of a human aspect exists as well. If the system which is regarded, includes human beings, which it often does, factors like alertness or fatigue will influence the system resilience [Str12]. Even aspects like the graphical user interface may be considered, since it may cause more frequent failures and therefore may be less often able to sustain required operations. Other metrics which target tolerable disturbances, may be metrics like the ability to return to the original state with k faulty components, communication up to a t-bit communication error or m erroneous inputs.

Chapter 3

Related work

This section describes related research concerning the topic of automated performance regression detection not limited to microservice architectures. Surprisingly, at the time of writing the existing research work on performance of microservice systems seems to be limited. The computer science bibliography dblp [Dbl] returns only 8 results for the query “microservice performance”. The related work on microservice architectures therefore is a short section. Nonetheless this chapter will give a short overview over the existing research work on microservice performance attributes. Afterwards it offers a detailed description of existing performance regression detection approaches.

3.1. Microservice performance research

Most similar to this work is the master thesis “Performance anomaly detection in microservice architectures under continuous change” by Düllmann [Dül17]. Although the title suggests that the research focus was nearly identical, the two works differ a lot. Düllman focused on observing the performance impacts of changes in a running microservice environment, while this work focuses on redeploying a test system for regression detection. In his work, he changed performance attributes of a single microservice during runtime. Opposed to that, the approach of this work focuses on observing the whole system and redeploying the whole system for every single change. Furthermore, Düllmann used architectural knowledge of the microservice system and focused mainly on how to prevent false positives triggered by high loads, which are observable during the deployment of a microservice. This work does not rely on architectural knowledge, although some of the approaches internally build a model of the relationship between the different metrics. Additionally, Düllmann used an artificial microservice system which consisted out of three microservice instances. This thesis evaluates an existing microservice platform which consists out of more than 20 microservice instances and has

to some amount a realistic use case. Finally, this work includes some general research focusing on the behavior of microservice metrics.

Gribaudo, Iacono, and Manini [GIM17] performed research on simulation-based estimation of performance attributes of microservice architectures. Their work mainly focused on infrastructure parameters of a microservice system which may influence performance. A simple example for such a parameter would be the number of available nodes in a cluster.

In a general survey, presented different problems and possible research directions in the field of microservices. They argue that special solutions for performance regression detection have to be investigated. Although this thesis's work evaluates the possibilities of common techniques and researches their possibilities and challenges. It shows their performance and highlights existing challenges.

The work of de Camargo et al. [Cam+16] focused on developing a testing framework for microservice systems. They tried to tackle test automation and the problem of keeping test specification consistent with rapidly changing microservices.

3.2. Existing performance regression detection approaches

There are several approaches to performance regression detection available. Since one of the main goals of this thesis is evaluating and investigating their usability for microservice architectures, the following sections will go into detail on how the different approaches work.

Inspired by the principles of a systematic literature review, the following approaches were found. The main goal of the research was to find a broad collection of performance regression detection approaches. To find those approaches, the two common research search engines dblp and Google Scholar, were used. As initial search queries "performance regression detection" and "performance regression test" were chosen. Out of those results, relevant papers which describe approaches for performance regression detection were selected. The related work sections of the selected papers, were used for finding further relevant research works.

3.2.1. Student t-test based performance regression detection

Statistical tests, such as the Student t-test are a commonly used approach in comparing measurements of two different versions of the same software system [Sha+15]. In a study, Shang et al. [Sha+15] propose a new approach to performance regression

Table 3.1.: Overview Student t-test

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
Several runs till reaching confidence threshold	Yes	No*	1 step	No	Very common	No*

detection. They compare their results, which will be presented in Section 3.2.2, to classical statistical hypothesis tests such as the Student t-test [Sha+15]. They conclude that such tests do not perform well and lead to a high number of false positives. Although their research shows that the quality of results depends highly on the chosen SUT, they conclude that the overall performance is not very promising.

3.2.2. Regression models on clustered performance counters

In the above mentioned work, Shang et al. [Sha+15] propose an approach which uses regression models on clustered performance counters. Regression models are models which allow to make predictions about dependent variables, given the values of some other explanatory variables. Such models are built out of sets of data points. There are different types of regression models. For example, linear regression models try to model the relationship of the data set by use of a linear function.

Clustering is a method which allows bundling of similar datasets. Since software systems can have thousands of performance counters, it is important to do a selection of metrics which shall be regarded. Especially when working with a model-based approach it would be infeasible to build a model for every counter.

Shang et al. therefore eliminate performance counters which can be considered redundant. In terms of redundancy analysis they use R^2 to step-wise eliminate performance counters over a predefined threshold. In a second step, Shang et al. cluster the performance counters, to reduce the models to be build. They use an n-dimensional representation for the performance counters. Each dimension represents a time step / slice of the load test and holds the corresponding value of the performance counter. By use of the Pearson distance metric, a hierarchical clustering based on average cluster distance is performed. By use of the Calinski-Harabasz stopping rule the clusters are split into separate clusters. Out of every cluster, the performance counter which deviates the most between the two software versions, is selected. Deviation is measured by use of a Kolmogorov-Smirnov test. For this counter, a linear regression model with the remaining counters as independent variables is build. Finally, for the performance regression detection, the model which was build out of the performance counters of an earlier version is used to predict the expected value for the selected performance counter of every cluster. This prediction is compared to the real values and the average prediction error is calculated. If the maximum of all average prediction errors, is higher than a given threshold, a performance regression alert is issued. Figure 3.1 visualizes

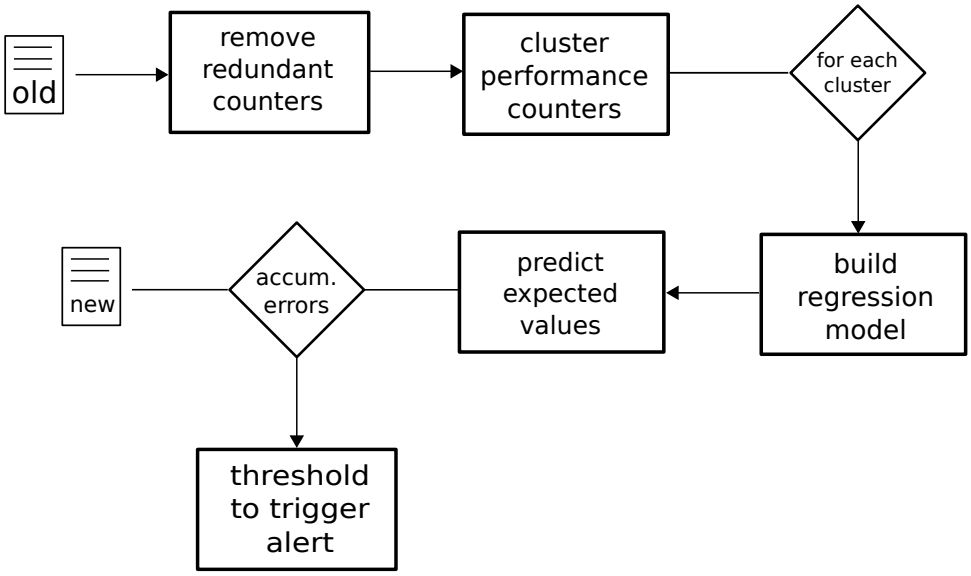

Figure 3.1.: Visualization of the process of regression models on clustered performance counters regression detection

the general process of the approach.

The results of the approach are promising. The proposed approach leads to better results than pure statistical Student t-tests or manual selection of variables for regression models.

Table 3.2.: Overview regression models on clustered performance counters

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
One run	Yes	-	6 steps	No	-	No*

Since there is no direct need for two separate test runs, the duration of this detection approach is smaller than the classical approach of two separate runs to make the loads comparable. This approach could be used for microservice performance metrics such as scalability, elasticity, and resilience as well.

3.2.3. Signature-based performance regression detection

Malik, Hemmati, and Hassan [MHH13][Mal10] propose an approach which is similar to the one of Shang et al. [Sha+15]. They focus on the reduction of relevant datasets as well. Malik et al. call the resulting small sets of performance counters with its values signatures. Their performance regression detection focuses on comparing such signatures.

Instead of clustering, they make use of principal component analysis (PCA). Principal component analysis is a statistical procedure which projects an n-dimensional dataset to a q-dimensional dataset with $q < n$. The results of the PCA are principal components (PC). PCA keeps the information loss of this projection low. Malik et al. suggest that PCA scales better than clustering algorithms. Afterwards they select a subset of PCs which represents 90% cumulative variability. This means that the selected PCs explain at least 90% of the variability of the collected measurements.

Afterwards they do a “PC decomposition” with the goal of extracting the most relevant performance counters out of the PCs. The exact technique of “PC decomposition” is not described in detail and there is no common technique with that name. For a possible reimplementation in later parts of this work, “PC decomposition” is understood as using the values of the linear combinations for the single PCs and metrics. For extracting the

most relevant performance counters out of this decomposition, they calculate weights for each performance counter and principal component and select the most relevant ones by use of a tunable threshold. The resulting weights of performance counters of a baseline test, are compared to the weights of the same performance counters in the newer version of the system. A deviation in weights suggests, that the distribution of the performance counter values have changed between the two versions. If the deviation crosses a certain threshold, a performance regression alert is issued.

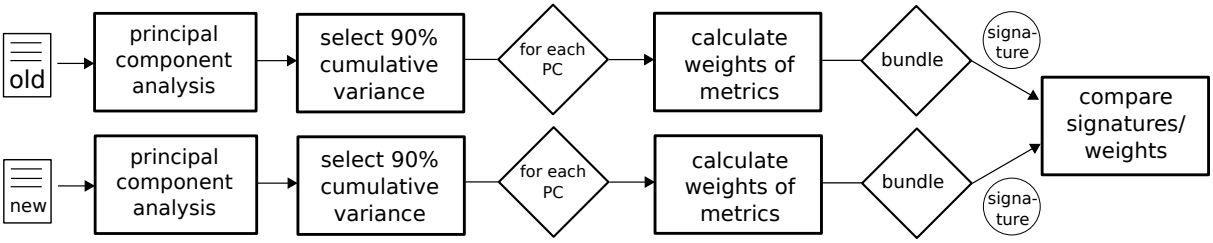

Figure 3.2.: Visualization of the process of performance signature-based regression detection

Figure 3.2 visualizes the general process of the approach.

Malik et al. propose another approach, which is supervised. This means, that manual inspection and labeling of data is needed. Although the labeling is only needed for the performance counters of the baseline tests, this approach is not considered in this paper. There are two reasons for this:

1) Malik et al. conclude that it is rarely possible for performance engineers to do such a labeling and

2) the alternative approach of PCA does only perform slightly worse.

Since the signature-based approach does not have specific requirements towards the used performance measurements, it is expected to be easily adapted for microservice performance metrics. Nonetheless, it is unclear whether the microservice performance metrics would be excluded in the performance counter reduction phase.

A big advantage of this approach is that it focuses on large scale software systems and therefore is expected to perform well in terms of scalability to big software systems. Additionally, Malik et al. suggest that the PCA approach performs better than clustering approaches. A direct comparison to the approach of Shang et al. [Sha+15] may not be possible, since Malik et al. use k-means clustering as a reference opposed of the hierarchical clustering of Shang et al. .

In terms of disadvantages, it's worth mentioning that the given approach offers no direct solution for changes in load and therefore would need the common double set of load

Table 3.3.: Overview signature-based performance regression detection

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
40+ samples	Yes	No*	3 steps	No	-	No*

tests to assure comparability. Furthermore Malik et al. found out, that their approach needs at least a base of 40 samples to work. Although they suggest that higher sampling rates easily tackle the problem, it is an additional requirement which other approaches do not have.

3.2.4. Performance regression detection with transaction profiles

Ghaith et al. [Gha+13] proposed a new approach for performance regression detection and did further research on it which was published later [Gha+16]. Their approach is based on so-called transaction profiles. Transactions are interactions of a user with the software system to invoke various application functions (e.g., login, browsing catalogs, etc.). In contrast to the common load metrics of total response time for such transactions, the approach of Ghaith et al. focuses on calculating the load-independent transaction profile for a transaction. A transaction profile describes which resource utilization a single transaction request produces (e.g., CPU usage on node 1, memory usage on node 2, etc.). The associated value to the whole transaction profile is the total time needed for one single transaction of this profile. Since such a transaction profile therefore describes the load of a single request, such transaction profiles are considered to be load-independent. In common setups, often two separate load tests are performed for performance regression detection to be able to compare the measurements made in different loads. One test is performed under the same load as the previous one, to be able to compare the results of both tests in performance regression detection. The other test is performed under the load which fits the current productive setup and loads. It is used for capacity analysis or future regression detection. Ghaith et al. report that such a

load independent metric makes it possible to eliminate the first of those two test runs, since the load independent metrics are directly comparable to their earlier values. This leads to a reduction of load testing duration of up to 50% [Gha+16].

For calculating the transaction profile values, three input parameters are needed. The first two, response times and resource utilization are common metrics for load testing setups and have good tooling support. The third one is a Queuing Model Network (QNM) of the SUT. Building and validating a QNM must be performed manually for complex systems. After an initially building a QNM, it can be used and updated for later runs. Furthermore, QNMs of typical deployment topologies exist in research and can support building the QNM.

Finally, the QNM has to be what Ghaith et al. call reverse-solved. Although there is no analytical solution for this process, search-based approaches for approximating the result are applicable. To optimize the duration of such search-based approaches, the transaction profile of earlier runs may be used as initial starting points.

Case studies of Ghaith et al. show that transaction profiles can be considered a more load stable metric than total response time, although the transaction profile values still are impacted by different load levels. Especially loads which lead to high levels of software contention, are impacting the transaction profile values. For performance regression detection based on transaction profiles, such loads must be avoided.

Additional inaccuracies in the method result out of ignoring network delays and usage of the BCMP approximation for QNMs. The BCMP is an often-used class of queuing networks, which does not exactly fit to realistic setups because it uses concepts such as infinite queues for servers or the estimation of equal service time for every customer.

For the performance regression detection Ghaith et al. use thresholds calculated by test runs on software versions with so called performance-safe changes. Those changes are considered to be of no performance impact. They use a 95% percentile of the deviations for each transaction type.

3.2.5. Statistical process control techniques using machine learning

Nguyen et al. [Ngu+12] propose an approach to performance regression detection which is backed by control charts. Control charts are commonly used in statistical quality control of manufacturing processes. Control charts show whether a process deviates from an earlier baseline of the same process. To use control charts, two requirements have to be fulfilled. First, the input parameters of the process should be considerable constant and secondly, the process output should be of a normal distribution. When trying to use control charts in software performance regression detection, those two requirements are

Table 3.4.: Overview transactional profiles

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
One run	Yes	Yes	4 steps	No	-	No

Table 3.5.: Overview statistical process control techniques using machine learning

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
One run	Yes	Yes	3 steps	Yes	Use of common technique	No*

not fulfilled. The input parameters of the process, which resembles the load of the SUT, are not constant. This is the case because load tests use randomizers for load generation, but even more importantly, load should be adaptable to real production loads, which change over time. Nguyen et al. propose to use machine learning techniques to learn parameters of a linear scaling factor to make performance metrics of different loads comparable. Their assumption is, that in a well-structured system performance metrics stand in a linear relationship to load. Their evaluation suggests, that this procedure is very accurate. The second requirement, a normal distribution of output parameters, does not hold without adaption either. Furthermore, their research shows that most of the distributions are bi-modal. They explain this with bookkeeping tasks which the system performs in idle states. Because of those bookkeeping tasks, the normal distribution does not drop to zero on the left-hand side. Nguyen et al. propose to search for the local minimum between real load and bookkeeping tasks load in the distribution and eliminate all loads to the left of it. This filtering leads to a highly improved fitting to normal distributions in their measurements. Figure 3.3 gives a visual aid for explaining the filtering approach. The red line shows the original distribution. Some small spikes on the left-hand side in the lower area of the distribution are cut away by removing the first maximum. All measurements with values lower than 70 are removed. The remaining distribution (blue dotted line) fits better to a normal distribution than the original red one.
The approach of Nguyen et al. stores all results of performance regression tests in one repository as baseline tests. The performance regression detection therefore does not have to rely on a single reference version.

Figure 3.4 visualizes the general process of the approach.

This approach has some requirements concerning the used metrics. They have to be of a normal distribution, or at least have to be filterable to reach a normal distribution and they have to be in a linear relationship to load. Because of the adaption of the performance metric values to fit to different loads, a double execution of load tests to make their results comparable under different loads is not necessary.

Nguyen et al. see their biggest advantage compared to other approaches in the simplicity and intuitiveness of the approach.

3.2.6. Performance regression unit testing

The approach of Horký et al. [Hor+13] focuses on how performance tests can be integrated into the software development life cycle from the very beginning of development. They argue that at early stages of development load testing can not depend on the

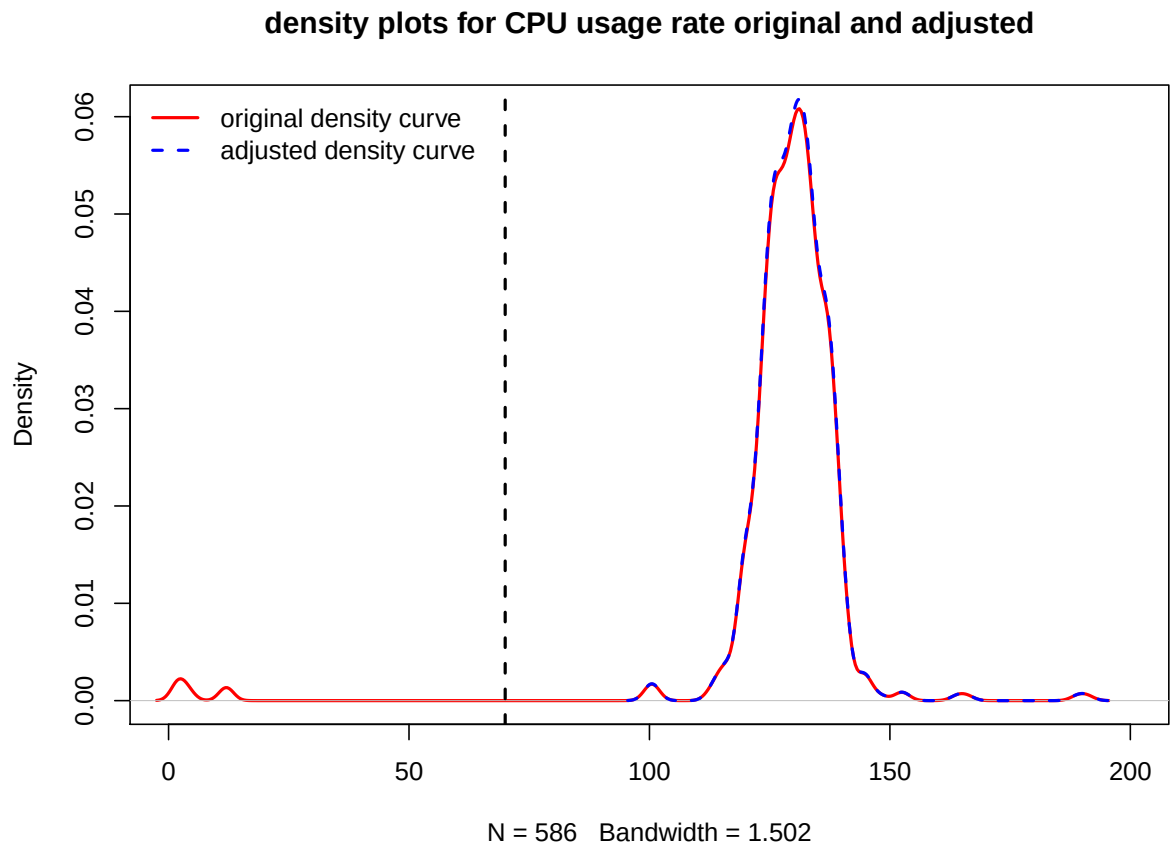

Figure 3.3.: Visualization of the proposed filtering technique of Nguyen et al.

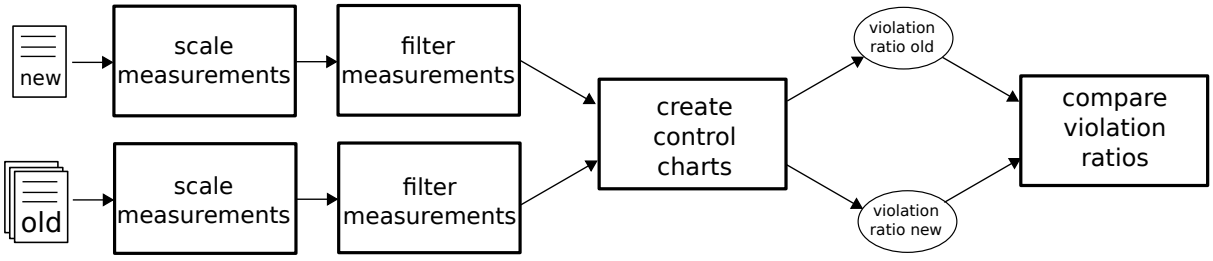

Figure 3.4.: Visualization of the process of statistical process control-based regression detection

availability of large components and therefore propose an approach which they call performance unit testing. Unit testing is a technique which is very common in functional tests. In functional tests, unit tests allow to test even small code segments based on example data calculated by use of the specification. Performance unit testing is not as easy as functional unit testing. Most of the time there is no specification of performance requirements on a low level, which makes verifying fulfillment of those impossible. Furthermore, performance is hardware dependent which makes a naive comparison of execution times senseless. Horký et al. therefore propose to use so-called baseline functions as reference. For example, a baseline function for a sorting algorithm could be the Java Array sort implementation. In terms of performance regression such a baseline function could be an earlier version of the method which was considered to be of sufficient performance. With Stochastic Performance Logic (SPL), a first order logic for performance comparisons, Horký et al. introduced a way of specifying performance relations between methods. The evaluation of such a SPL formula is done by statistical hypothesis testing.

The formulas can be annotated directly into Java code. The tooling support is surprisingly good if only Java is considered. A command line interface for test execution, automated HTML report generation, an Eclipse plugin as well as Git and SVN integration for automated pulls of older project versions as baseline are available [Dev].

The performance regression detection in this approach is done with the Welch's t-test [Wel47], although the needed assumption of two sets of independent observations of random variables with the same distribution does not hold. Horký et al. conclude that the test is none the less usable, but test repetitions of around tens of thousands are needed. Furthermore, because of the non-deterministic behavior of load tests, they propose a 5% threshold for regression detection to avoid false positives. In their case study, test durations of 27 minutes for a code coverage of 18% are needed. To reach better test durations, they propose caching of earlier results to prevent duplication of performance tests.

Opposed to all other approaches of this work, this approach focuses on a unit level. Most commonly regression detection is performed on a component, integration or system level. To reach a system level evaluation with performance unit tests, all single components would need a performance requirement specification and performance unit tests. It is hard to imagine in a microservice environment because there the spectrum of used languages and tools is very broad.

Table 3.6.: Overview performance regression unit tests

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
Thousands of function runs	No	Yes	Impl. available	No	Inspired by common technique	?

3.2.7. Differential flame graphs

Bezemer, Pouwelse, and Gregg [BPG15] propose a visualization approach called differential flame graphs for supporting performance regression detection. Flame graphs visualize how much time a program spends in a certain stack trace during test execution. The visualization shows the aggregated time on the x-axis and visualizes the given stack traces on the y-axis. A software performance engineer can use such a flame graph for developing a better understanding of how much time given functions need for execution and on which layers of a function call was spent which amount of time. Out of the flame graphs of two distinct versions of the system, Bezemer et al. propose to build a differential flame graph. The differential flame graph uses color to show whether and how much a function's performance metric increased or decreased. Furthermore, a differential flame graph can be built upon the differences of the two flame graphs, so that the resulting differential flame graph only visualizes stack traces of function calls which changed their performance behavior between the two versions of the software. This approach is usable for visualizing stack based metrics, such as stack traces of execution. In an earlier paper Bezemer et al. [Bez+14] did research on how I/O usage could be collected as a stack based metric.

Although this approach is meant to support a manual performance regression detection process, an automated implementation is imaginable. A big issue concerning this approach is the availability of stack based metrics. Especially considering a microservice

Table 3.7.: Overview differential flame graphs

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
One run	No	No*	Impl. available	No	Use of common technique	No

environment, the majority of metrics is not based on stack traces. In their paper, Bezemer et al. mention that differential graphs may as well be used in graphical user interface evaluation. A stack-trace equivalent would be the navigation of the user through the user interface. For microservices, an interesting approach could be using the traces of a request through the microservice architecture. A possible differential flame graph would show how a request uses different microservices and how much time the single interactions need.

3.2.8. Mining performance regression testing repositories

Foo et al. [Foo+10] propose to use data from earlier performance tests to extract association rules, which show relations between different performance counters. Figure 3.6 gives examples for possible rules. On the left hand side of each rule are zero to n observations, which lead to 1 to m resulting observations.

Foo et al. [Foo+10] suggest to detect performance regressions by testing those association rules of earlier runs against the data of a newer run. If the confidence in a rule deviates more than a given threshold, a performance regression alert is issued. In a first step of their approach, the data of the performance metrics is converted into cardinal data (low, medium, high). Foo et al. afterwards extract the association rules by use of data mining concepts on this cardinal data set. They find frequent data sets and association rules by use of the apriori algorithm which returns support and confidence

values for possible association rules. These confidence levels are compared between different versions of a software system, to detect performance regressions.

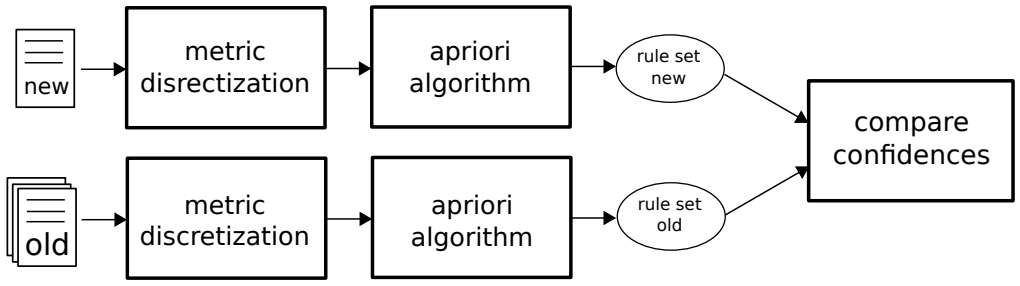

Figure 3.5.: Visualization of the process of mining performance regression testing repositories regression detection

Figure 3.5 visualizes the general process of the approach.

The results of the approach seem to be comparable to others. The flexibility concerning use of different metrics seems to be promising for usage in a microservice environment. The need for a big repository of existing performance load test data may be a disadvantage of the approach.

$$\{\text{carts network/rx\_rate high}\} \rightarrow \{\text{carts cpu/usage\_rate high}\}$$

$$\{\text{carts cpu/usage\_rate high}\} \rightarrow \{\text{carts network/tx\_rate high, carts-db network/rx\_rate high}\}$$

$$\{\} \rightarrow \{\text{carts memory/usage medium}\}$$

Figure 3.6.: Examples for association rules

Table 3.8.: Overview mining performance regression testing repositories

Speed of detection	*Support of metrics*	*Adapt. to changing loads*	*Complexity*	*Load testing repository*	*Commonness*	*Change in distribution*
One run	Yes	No*	3 steps	Yes	-	No

Chapter 4

Comparison and implementation of approaches

After giving a short overview over the different approaches of performance regression detection, this thesis first compares the presented approaches to performance regression detection. In a second step, a subset of the approaches is chosen to be implemented and evaluated in a microservice environment. Criteria by which the approaches are compared are introduced.

A second chapter describes how the selected approaches were implemented.

4.1. Selection criteria

This thesis strives to research performance regression detection approaches which are especially promising for microservice architectures. Therefore, the selection focuses on criteria which are relevant to the microservice environment.

The main aspects, which are expected to be relevant for performance regression detection in a microservice environment are:

Speed of detection Since microservice architectures are rapidly changing environments, one of the biggest issues in integrating performance tests into the delivery pipeline, is the duration of the overall detection time. Different approaches vary in the number of test runs needed (e.g., two runs with different loads) or the number of data points needed for a reliable detection.

Support of different metrics/Adaptability to microservice metrics Since the selected approaches should be able to be enriched with microservice performance metrics like scalability, elasticity and resilience, it is important that the approach would generally support use of such metrics.

Adaptability to changing load Not only the overall system is rapidly changing in microservice architectures, the load of the system under test must be adapted over time as well to fit the load of the productive system. Approaches which have some possibility of adapting to changing loads are better suited than those who do not.

Complexity The complexity of the selected approaches is relevant out of two reasons. First, a complicated approach is hard to understand and results may be complex in evaluation and communication. Secondly since the extent of this thesis is limited, approaches which offer an existing implementation or are easier to implement are more likely to be chosen than more complex ones. Since there is no direct way of measuring the complexity of an approach, the comparison is done by estimating the number of different steps which have to be performed to perform the regression detection. This estimation is not precise and to some extend influenced by personal experience and subjective opinion.

Need for a load testing result repository Some approaches need a repository of old load testing results. Since this is considered to be an additional effort for software performance engineers, such a requirement is considered to be a disadvantage.

Commonness Performance regression detection approaches which are currently more common than others, are considered to be of more interest in research. Not only are findings concerning such approaches of common interest, but they may offer an opportunity to help establishing such an approach in microservice environments.

4.2. Comparison of approaches

Table 4.1 shows a tabular comparison between the approaches for performance regression detection. The first row, speed of detection, gives an overview what effort in terms of performance testing is needed for starting the regression detection.

The second row, support of metrics, expresses how easily the given approaches are adaptable to new or different metrics.

Unit testing is considered to be not adaptable, since the tests are performed on a function level, where interesting aspects such as inter process communication and scalability are not directly visible. Since differential flame graphs are only usable for datasets which are stack-based, this approach is only usable for few metrics.

The third row, adaptability to changing loads, expresses whether the algorithm of the approach includes mechanisms to deal with changing loads when comparing the current system with an older version. The value “No*” indicates that although the approach does not tackle the issue, the solution of repeating the new test under the old load set up is possible. Since this approach doubles the load testing time, it is considered to be a disadvantage.

The forth row, complexity, tries to grasp the complexity of the given approach. Since such a comparison is not possible in general, the comparison is done by estimating the steps needed to execute the algorithm. As an example: Considering the approach of signature-based performance regression detection (Section 3.2.3), one must do a principle component analysis, afterwards a principle component decomposition and finally do a comparison of the retrieved performance signatures. Overall those are considered 3 separate steps. As said, this comparison is only chosen because of a lack of better ways to compare complexity.

The fifth row, load testing repository, expresses whether the given approach needs a repository of old load tests. The approach of statistical process control (Section 3.2.5) needs a repository for machine learning of the α and β values of the linear models. Mining Repositories (Section 3.2.8) needs a repository to build the association rules.

The sixth row, commonness, describes how common the approaches are in comparison of usual techniques of software performance engineering. The Student t-test (Section 3.2.1) is a very common approach [Sha+15]. The statistical process control approach (Section 3.2.5) uses control charts, which are commonly used charts in process control. The unit testing approach (Section 3.2.6) is inspired by common functional unit tests. The differential flame graph approach (Section 3.2.7) uses the, according to Bezemer et al. [BPG15], common visualization of flame graphs.

The last row, change in distribution, shows whether the approach is sensitive to changes in the distribution of the collected data sets. “No*” means, that the algorithm itself does not do a distribution change detection, but when using the right metrics, like variance of total response time, a detection of distribution change is possible.

Table 4.1.: Tabular comparison between approaches

Approach	**Student T-Test**	**Clustered counters**	**Signature-based**	**Transactional profiles**	**Statistical process control**	**Unit testing**	**Differential flame graphs**	**Mining repositories**
Description	3.2.1	3.2.2	3.2.3	3.2.4	3.2.5	3.2.6	3.2.7	3.2.8
Speed of detection	Several runs till confidence threshold	One run	40+ samples	One run	One run	Thousands of function runs	One run	One run
Support of metrics	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Adaptability to changing loads	No*	-	No*	Yes	Yes	Yes	No*	No*
Complexity	1 step	6 steps	3 steps	4 steps	3 steps	Impl. available	Impl. available	3 steps
Load testing repository	No	No	No	No	Yes	No	No	Yes
Commonness	Very common	-	-	-	Use of common technique	Inspired by common technique	use of common technique	-

4.3. Selection of approaches

Table 4.1 gives a tabular overview over the aspects of the different approaches.

In this thesis, the approach of performance unit testing and differential flame graphs will not be further researched. Performance unit testing does not fit the targeted testing scenario because it tests on a unit level. Although regression testing can be performed on unit level, the more common use case is to perform it on a system or component level. Furthermore, use of different tools and languages is very common in microservice environments. Proper implementation of performance unit tests would therefore be nearly impossible since they depend highly on the chosen languages. The existing research concerning scalability, elasticity, and resilience offers no possibilities of collecting those metrics on a unit level.

Differential flame graphs will not be further considered in this thesis because their use case, especially in terms of metrics, is too narrow. Furthermore, it is unclear how an automated regression detection approach would be realized with those graphs. Additionally, future applications of including measurements w.r.t. scalability, elasticity, and resilience are not applicable. Differential flame graphs could nonetheless be a promising visualization for traces throughout the microservice system.

Although offering a performance boost by trying to perform load independent tests, the technique of transaction profiles is not further looked into. The main reasons are that the implementation of the approach seems quite time consuming. Additionally, the results of the research work suggested that the results were only to a limited degree load independent. Furthermore, the need for a queuing model of the specific system under test is an additional burden that is non-trivial to realize for a microservice architecture. Networking delays were not considered in the original approach. Since network traffic is one important key aspect of microservice architectures, this might be more important than first expected. Lastly it is unclear how well and how fast a search-based reverse solving of a queuing network could be performed.

The Student t-test, the signature based approach and the statistical process control approach will be suspect of a practical evaluation.

The Student t-test is chosen because it is very common and simple to implement. It is therefore a good choice to evaluate other approaches against. This technique is often used although it is expected that not all performance metrics are normal distributed and it therefore should not be applicable or handled with care.

The approaches of “Regression models on clustered performance counters” and “Signature-based performance regression detection” are considered to be similiar. Both approaches focus on minimizing the number of relevant counters to compare. Malik

[Mal10] directly compares his PCA based algorithm to a cluster based algorithm and concludes that the PCA based algorithm outperformed a clustering approach. Although Shang et al. [Sha+15] do not stop after clustering but do further work by building regression models on top of them, together with the fact that [Mal10] promises to be easier to implement, Signature-based performance regression detection" is chosen while "Regression models on clustered performance counters' is not being implemented.

The approach of Nguyen et al. [Ngu+12] is interesting for two reasons. First of all, they propose a technique to turn most of the non-normal distributed measurements into normal distributed measurements. This could be interesting for the Student t-test as well. Secondly by choosing control charts as basic technique they use a very common approach known in software performance monitoring.

Last but not least, the approach of Foo et al. [Foo+10] is chosen because of its simplicity in implementation and because by sticking to very basic ordinal versions of the data set, it is very special in its way of approaching the problem of performance regression detection. It is expected that because of its nature it should perform differently and it may be able to detect different kinds of regressions.

4.4. Implementation of approaches

For all selected approaches, the basic setup shown in Figure 5.2 was chosen. A local mirror of the InfluxDB, a Java implementation and an R server were used. The main work of the implementation was implementing the approaches in Java. Since all approaches use similar functionalities, first a general framework for performance regression analysis was written. The key features of it are connecting to the InfluxDB by use of the REST API, extracting single test runs out of the raw measurements, normalizing the recorded measurements as described in Section 5.2.2, offering a connection to the R server and evaluation functionalities to later evaluate the set of runs with regressions and without. As described later, the R server offers a set of different statistical and numerical techniques which are well documented and tested. It helps avoiding implementation errors by building upon well tested structures.

The following sections give a short description of the implemented approaches as well as an algorithmic pseudo code representation. All algorithms share the same basic interface of DetectRegressions($repo$, $test$). $repo$ is the representation of the load testing repository containing all past load tests and $test$ contains the performance measurements of the new test run on which the regression detection should be performed. Those two data structures should be imagined like databases where the corresponding fields link to corresponding datasets. The reason for this analogy is that in the real implementation

$repo$ and $test$ indeed are kept in the InfluxDB. For example $repo.pods$ returns a set of representations of all the pods which were observed in the load testing repository. $pod.metrics$ for example would return a set of all metrics which were collected for a certain pod.

4.4.1. Student t-test based performance regression detection

The Student t-test was the easiest to implement. Since there was no reference paper describing a performance regression detection approach with the Student t-test, the approach was implemented in a way which would be expected in a performance engineering use-case. Algorithm 4.1 shows the algorithm used for performance regression detection with the Student t-test. The algorithm basically describes that for every singe pod and metric a Student t-test is performed (line 7) and if the p value is lower than 0.05 a regression is reported (line 9). For the Student t-test and calculating means the R implementations was used.

Algorithm 4.1 Student t-test regression detection pseudo code

```
 1: procedure DETECTREGRESSIONS(repo, test)
 2:     for all pod ∈ repo.pods do
 3:         for all metric ∈ pods.metrics do
 4:             allOldData ← repo.LASTMEASUREMENTSOF(pod,metric)
 5:             allNewData ← test. ALLMEASUREMENTSOF(pod,metric)
 6:             if allOldData.variance ≠ 0 then
 7:                 pValue ←TTEST(allOldData.mean,allNewData)
 8:                 if pValue ≤ 0.05 then
 9:                     REPORTREGRESSION(pod,metric)
10:                 end if
11:             end if
12:         end for
13:     end for
14: end procedure
```

4.4.2. Statistical process control techniques using machine learning

The machine learning aspect of this approach was neglected in the evaluation, since it is not directly associated to the detection approach. For sake of reducing evaluation complexity, no different loads were used in the observation. Therefore, the machine learning part of the approach should not be relevant.

Algorithm 4.2 shows the used approach in pseudo code. For every pod and metric, the algorithm performs an evaluation of the control chart violation ratios. If the violation ratio increased, a performance regression is reported. The lines 6 to 14 may be irritating. They implement the filtering approach to get more normal distributed data. The filtering is only performed if the p value of the Shapiro Wilk test increased by doing so. The R implementation was used for performing the Shapiro Wilk test and calculating means and standard deviations.

4.4.3. Signature-based performance regression detection

The signature-based performance regression detection approach extracts performance signatures by use of principal component analysis (PCA). Algorithm 4.3 shows the pseudo code implementation of the algorithm. Basically, the performance signature of an old and a new test are generated (line 2 to 5) and afterwards compared (line 6 to 15). The signature extraction applies PCA (line 20) to a matrix of the performance measurements. In this matrix, the single rows represent different measurements and the different columns represent different metrics. To apply the PCA, zero variance measurements have to be removed first. This step is left out in the pseudo code to make understanding the code easier. The final signature is build out of the first x principal components which have an accumulated variance of 90% of the original data set. For every principal component the weights of the metrics are afterwards extracted out of the corresponding fields of the eigenvector. The eigenvector represents the linear combination of the metrics that results in the certain principal component.

4.4.4. Mining performance regression testing repositories

The approach of mining performance regression testing repositories uses the apriori algorithm to extract association rules out of the measurements. Algorithm 4.4 shows a pseudo code implementation of the approach. After extracting the association rules (line 2 to 3) the confidence change is calculated. If it is higher than a certain threshold (0.02 in this example) a performance regression is reported.

For performing the apriori algorithm the measurements first have to be transformed into a solely ordinal type of data (line 13 to 27). Afterwards the resulting data set is put into the apriori algorithm. The apriori algorithm extracts all rules which have a minimum support and confidence level. These two thresholds are adjustable but not shown in this pseudo code representation.

Algorithm 4.2 Statistical process control techniques regression detection pseudo code

```
1: procedure DETECTREGRESSIONS(repo, test)
2:     for all pod ∈ repo.pods do
3:         for all metric ∈ pods.metrics do
4:             allOldData ← repo.ALLMEASUREMENTSOF(pod,metric)
5:             allNewData ← test. ALLMEASUREMENTSOF(pod,metric)
6:             pValueOrig ←SHAPIROWILKTEST(allOldData).pValue
7:             if pValueOrig ≤ 0.05 then
8:                 allOldDataFiltered, cutValue ←REMOVEFIRSTPEAK(allOldData)
9:                 pValueFiltered ←SHAPIROWILKTEST(allOldDataFiltered).pValue
10:                if pValueFiltered > pValueOrig then
11:                    allOldData ← allOldDataFiltered
12:                    allNewData ← {x ∈ allNewData : x > cutValue}
13:                end if
14:            end if
15:            bVio ←GETVIOLATION(allOldData,allOldData)
16:            tVio ←GETVIOLATION(allNewData,allOldData)
17:            if tVio > bVio then
18:                REPORTREGRESSION(pod,metric)
19:            end if
20:        end for
21:    end for
22: end procedure
23:
24: procedure GETVIOLATION(testData, refDat)
25:    total ← 0
26:    violation ← 0
27:    for all val ∈ testData do
28:        total ← total + 1
29:        if val ∉ [refDat.mean − 3 × refDat.sd, refDat.mean + 3 × refDat.sd] then
30:            violation ← violation + 1
31:        end if
32:    end for
       return violation / total
33: end procedure
```

Algorithm 4.3 Signature-based performance regression detection pseudo code

```
1: procedure DETECTREGRESSIONS(repo, test)
2:     lastOldRun ← repo.last
3:     lastNewRun ← test.last
4:     oldSignature ←EXTRACTSIGNATURE(lastOldRun)
5:     newSignature ←EXTRACTSIGNATURE(lastNewRun)
6:     for all x ∈ newSignature.range do
7:         for all metric ∈ newSignature.pcs[x].metrics do
8:             if x ∈ oldSignature.range ∧ oldSignature.pcs[x].contains(metric) then
9:                 if abs(newSignature.pcs[x][metric] − oldSignature.pcs[x][metric]) >
   0.02 then REPORTREGRESSION('signature changed significantly ',metric)
10:                end if
11:            else if newSignature.pcs[x][metric] > 0.02 then
12:                REPORTREGRESSION('signature has new significant metric',metric)
13:            end if
14:        end for
15:    end for
16: end procedure
17:
18: procedure EXTRACTSIGNATURE(run)
19:    measurementMatrix ←GETMEASUREMENTMATRIX(run)
20:    pcaRes ←PCA(measurementMatrix)
21:    cummulativeVariance ← 0
22:    result ← {}
23:    for all pc ∈ pcaRes.pcs do
24:        if cummulativeVariance < 0.9 then
25:            cummulativeVariance ← cummulativeVariance + pc.var
26:            result.add(pc)
27:            for all metric ∈ run.metrics do
28:                result.lastPc[metric] ← result.lastPc.eigen[metric.numb]
29:            end for
30:        else
31:            break
32:        end if
33:    end forreturn result
34: end procedure
```

Algorithm 4.4 Mining performance regression testing repositories pseudo code

```
1:  procedure DETECTREGRESSIONS(repo, test)
2:      allOldRules ← repo.EXTRACTRULES(repo)
3:      allNewRules ← repo.EXTRACTRULES(test)
4:      for all x ∈ allOldRules, y ∈ allNewRules : x.type = y.type do
5:          confidenceChange ← 1 − (x.conf · y.conf + (1 − x.conf) · (1 − y.conf)) / (√(x.conf² + (1 − x.conf)²) · √(y.conf² + (1 − y.conf)²))
6:          if confidenceChange > 0.02 then
7:              REPORTREGRESSION(x,y)
8:          end if
9:      end for
10: end procedure
11:
12: procedure EXTRACTRULES(data)
13:     cardinalDataMap ← {}
14:     for all pod ∈ repo.pods do
15:         for all metric ∈ pods.metrics do
16:             data ← repo.ALLMEASUREMENTSOF(pod,metric)
17:             for all val ∈ data do
18:                 if val.value > data.mean + data.sd then
19:                     cardinalDataMap.put(val.time, metric + 'high')
20:                 else if val.value > data.mean − data.sd then
21:                     cardinalDataMap.put(val.time, metric + 'low')
22:                 else
23:                     cardinalDataMap.put(val.time, metric + 'medium')
24:                 end if
25:             end for
26:         end for
27:     end for
        return APRIORI(cardinalDataMap).rules
28: end procedure
```

Chapter 5

Evaluation

The following section introduces the main research questions of this thesis and presents the chosen methodology for answering them. By use of an empirical study, it presents an evaluation of the formulated research questions. It gives an in-depth explanation of the used setup, the system under test and the architecture of the performance regression detection prototype.

5.1. Evaluation goals

This section gives a short explanation of the research questions of this thesis. Since the main goal was researching the possibilities and challenges of automated performance regression detection in a microservice environment, two different focuses were set. The first one targets possible differences between common monolithic systems and the microservice environment and its metrics. Since most metrics of microservice architectures are collected on virtualized systems, different virtualization containers of one node may influence each other. The second focus is set on the concrete performance regression detection approaches and their performance in a microservice environment. The goal of this second focus is to evaluate how well current performance regression detection approaches perform in the new environment of microservices.

5.1.1. Evaluation of microservice performance metrics behavior

RQ1.1 Which metrics are available and commonly collected?

RQ1.2 How stable are metrics during a run?

RQ1.3 Can metrics be considered or adapted to be of normal distribution?

RQ1.4 How stable are metrics between system redeployments?

Since load testing measurements and their metrics are the foundation for performance regression detection, research on the overall behavior of typical metrics is performed. Main goals of evaluation concern the stability of those metrics. This stability will be evaluated during a single deployment as well as between several redeployments. If a metric has a high variance during a test run, long testing durations are needed to gather statistical significant results. If a metric does vary highly between redeployments, the values of such a metric during a single test run and deployment are not representative and several deployments and test runs may be needed to get significant results.

5.1.2. Evaluation of performance regression detection approaches

RQ2 How do the implemented approaches perform in a microservice environment?

The evaluation shall show advantages and disadvantages of existing approaches and shall evaluate their performance in a microservice setup. A perfect microservice performance regression detection approach would be expected to be efficient concerning needed load testing durations as well as the needed evaluation time. Furthermore, the results of the approach should have a high precision and recall. It should be simple to set up and use. Furthermore, the results of the regression detection approach should help the performance engineer in understanding what kind of regression and where it was found. The selected and implemented performance regression detection approaches will be evaluated in these categories.

5.2. Evaluation methodology

To answer the question of which metrics are commonly collected, the documentation of Kubernetes default monitoring tool Heapster was used as a reference.

The remaining questions were answered by performing an empirical study on a reference microservice system. To evaluate the research questions concerning the behavior of performance metrics in the context of microservice environments (Section 5.1.1), a series of 19 load tests, each 4 hours long, was performed in this test setup (Section 5.3). The SUT and load specification did not change in between the different load tests and redeployments, since the focus of the metrics behavior research is set on the stability during a run, in between runs and the distribution of the metrics' measurements.

A second set of runs was performed on the SUT to answer the remaining research question of how well the single performance regression detection approaches perform

in a microservice environment. To answer this question, a set of 44 load tests with durations of 2 hours each were performed. Out of the 44 test runs, 20 were performed without regressions and contained the unchanged system. Five of the 20 unchanged runs were used to simulate a performance test repository, which some approaches need. The remaining 15 runs without regressions were used to evaluate false positives of the performance regression detection approaches. Additionally, to the 20 unchanged runs, 24 runs were performed with injected regressions. For this purpose, 6 different kinds of regressions were injected (Section 5.3.6) and of each type of regression 4 load testing runs were executed. The load specification was not changed in between the single runs.

The final evaluation of the performance regression detection approaches was performed by testing every approach with each of the 15 regression-free and 24 regression-including versions of the SUT. For this final evaluation, all metrics which are described in Section 5.4, except for network and filesystem metrics were used. These metrics were excluded because they triggered a lot of false positive regression alerts.

5.2.1. Steady state detection

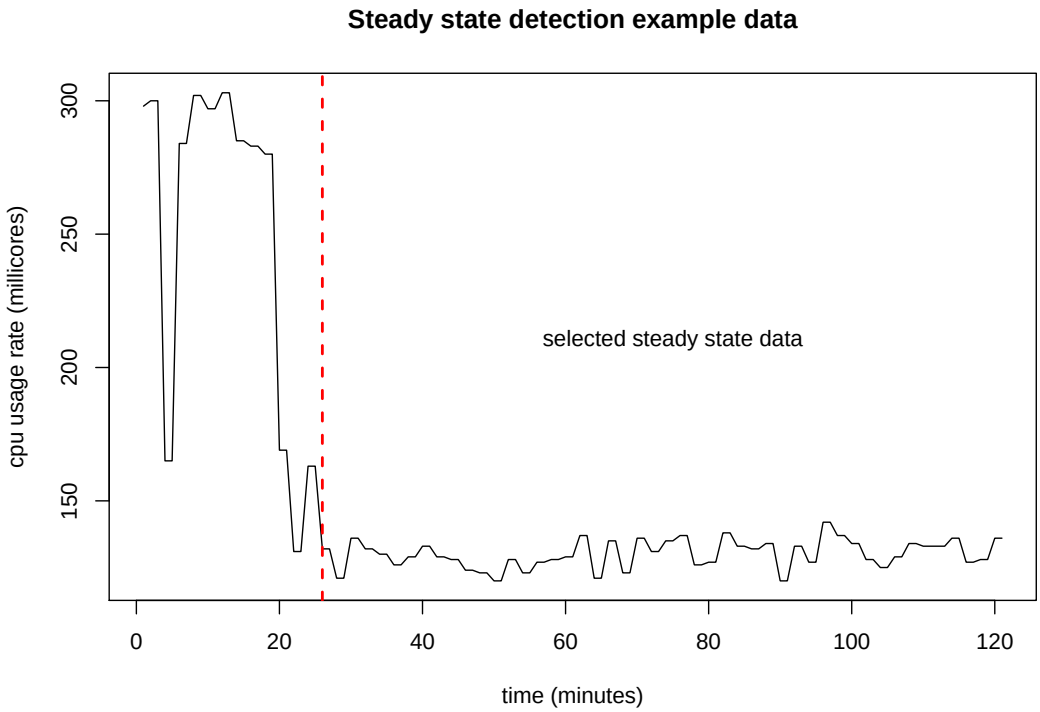

Figure 5.1.: Steady state detection visualization

When a SUT is deployed and put under load, it needs time to reach a so-called steady or stationary state. The steady state describes a state of the SUT in which the metrics can be considered stable. It is expected that in such a steady state, initial loads which may result from startup, caching and warming up of the overall system have reached a state in which they change negligibly. The difference between measurements influenced by early warm up and deployment tasks and steady state metrics is visible in Figure 5.1. The figure shows CPU usage rate (Table 5.3) of one test run. The red line marks the point after which the system can be considered to be in steady state.

An often-used naive approach for ignoring initial anomalies in metrics, is to ignore a fixed time span at the start of each test. Steady state detection is a possibility for speeding up the regression detection process since it allows to be more precise in removing samples. Therefore, a relevant amount of load testing is collected earlier.

In this thesis, the naive approach of ignoring a fixed time span at the beginning of load tests is being used. This approach was chosen since the steady state detection is not relevant for evaluating the performance between the selected approaches. For the sake of completeness, it is nonetheless worth mentioning it. One possible way to implement steady state detection is performing a trend analysis on the measurements while iteratively removing measurements from the beginning. When a point is reached, where no trend is observed, the remaining measurements can be considered to be in steady state. A more detailed description of possible ways to perform steady state detection is given by Shumway and Stoffer [SS06].

5.2.2. Metric normalization

Since microservice architectures are most commonly deployed in distributed setups, some challenges rise concerning the simultaneous collection and analysis of those. According to Foo et al. [Foo+10], there are mainly three issues to tackle:

Clock Skew Since the cluster is built out of different independent machines, the clocks of the different machines may not be identical. There may be offsets between them, which could lead to misinterpretations when comparing measurements performed with different clocks. Furthermore, different metrics may be observed at different rates, which leads to the fact, that at one certain point of time, there may be one metric available, but others are not.

Extended Test There may be several measurements collected directly after the test, where the system may not be under load anymore.

Delay The measurements may start collecting data at different offsets. This may be caused by different start up times or different resource utilizations on the different nodes.

These challenges were tackled in the testing environment by use of the following solutions:

Clock Skew In the test environment, measurements were only taken approximately every minute. The time offsets of the different clocks therefore should be neglectable, since no such time intervals are observed which are in the magnitude of common offsets. Furthermore, for later evaluation of the single metrics and measurements in the performance regression approaches, the measurements are first linearly interpolated in strict one-minute intervals, to allow for a best approximation of the real values of the different metrics at certain times.

Extended Test Similarly to ignoring data collected at the beginning of the test because of not steady states, a fixed offset at the end of the tests is ignored.

Delay Delays considering the collection of first measurements on the different nodes are not relevant, since the time to reach a steady state is longer than that of measurement collection startup. Since all measurements before reaching a steady state are ignored, this delay can be ignored.

5.3. Evaluation setup

The following section presents an overview over the systems which were used for the evaluation. It describes the architecture of the performance regression detection setup, the different software tools which were used and describes the cluster on which the evaluation was performed.

5.3.1. Overview

Figure 5.2 shows the overall testing and evaluation environment used in this thesis. Visualized as blue boxes on the left, one can see the system under test. In this thesis Sock Shop (Section 5.3.2), an artificial sock web shop, is used as a system under test. To observe the system in a busy state, several Locust (Section 5.3.4) instances simulate users putting the system under load. The Locust instances are visualized at the bottom of the figure. Heapster (Section 5.3.3) observes the resource consumption of the system and the single microservices. Heapster and the Locust load drivers store their collected data in a central InfluxDB (Section 5.3.3), which is visualized in the center of the figure.

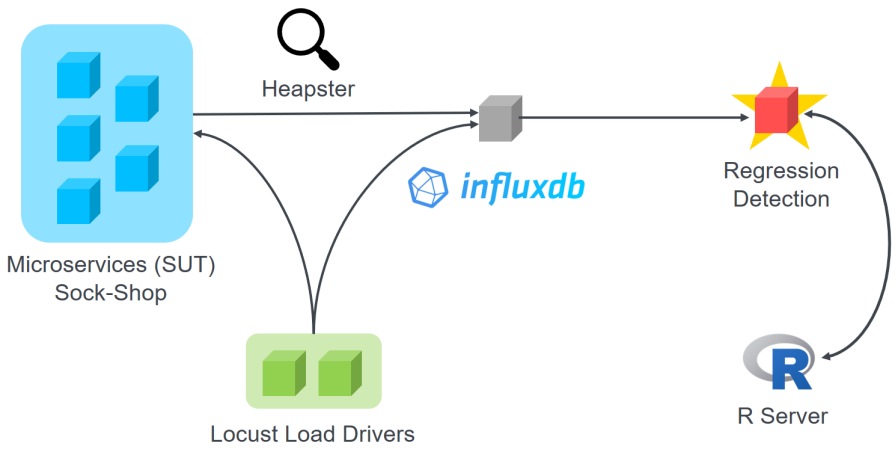

Figure 5.2.: Overall view on the test environment

Table 5.1.: Specification of the testing nodes

Image	Fedora-Cloud-Base-25-1.3.x86_64
CPU cores	4
CPU MHz	2300
RAM	8 GB
Disk space	80 GB

Which metrics are observed, is presented in detail in Section 5.4. The performance regression detection approach on the right of the figure, communicates with the InfluxDB and requests the information of the load test metrics. For statistical evaluation of the data sets, the implemented approaches use an R Server (Section 5.3.5), which is shown in the bottom right corner. The arrows in the figure visualize data flows between the single components and show how the components are connected.

Kubernetes (Section 5.3.3) is used for container management and deployment. It runs on an OpenStack cluster [Ope]. In the cluster, Kubernetes uses three virtual machines whose specifications are shown in Table 5.1.

5.3.2. Sock shop — a microservices demo application

In their work, Aderaldo et al. [Ade+17] research requirements for microservice research benchmarks. They research available candidates and evaluate whether they fulfill those requirements. Although Aderaldo et al. conclude that none of the available open-source candidates are mature enough to be used as a community-wide benchmark, they argue

that they already can be valuable for empirical research. Sock Shop [Soc], one of the proposed candidates, is a microservice demo application developed by Weaveworks, a company which focuses on cloud solutions. It consists out of 19 microservices which are implemented in Java, Go, and Node.js. Although the system is built as an artificial benchmarking platform, it resembles the back-end of a web shop which sells socks. Sock Shop was chosen as an evaluation platform for several reasons.

Documentation Sock Shop is quite well documented. Deployment instructions for the most common platforms are available. An architecture description exists and basic documentation for the single microservices is available.

Microservice architecture Sock Shop was designed with common architectural patterns in mind. Aderaldo et al. mention service discovery, database per service, and messaging as examples.

Load tests Sock Shop already offers extensive Locust [Loc] load testing scripts, which simulate users registering, logging in, browsing the catalogue, ordering socks and simulating even credit cards in checkout.

Autoscaling Sock Shop offers horizontal scaling scripts for Kubernetes [Kubb]. Therefore, one of the key aspects of the microservice architecture, high elasticity and automated adaption to load, is thoroughly represented.

Monitoring Zipkin is integrated in some of the microservices and collects traces throughout the system. A trace shows how a request propagates through the microservice system.

5.3.3. Kubernetes, Heapster, and InfluxDB

Kubernetes [Kubb] is an open-source system which offers functionality for deployment, orchestration, and scaling of microservice containers. Heapster [Heab] is an open source software which enables container cluster monitoring and performance analysis of the single containers. In the test setup of this thesis Heapster runs on the Kubernetes cluster and collects basic performance metrics such as CPU usage, memory usage, or network load on a container level. The collected data of the load testing tool Locust [Loc] and the data of Heapster are stored in an InfluxDB. InfluxDB [Inf] is an open source time series database. A time series database is a database which specializes in storing and querying data which has an ordered time dimension. Since the obtained metric measurements are essentially a tuple of time and value, it is sensible to use a time series database like InfluxDB in this setup.

5.3.4. Locust — a python load testing tool

Locust [Loc] is an open source load testing tool. It is used to put the SUT under a constant load. The load testing script of this thesis is inspired by the load testing scripts which Sock Shop is offering. To make sure that variance in recorded performance metrics is not a result of high variance in the load which was put on the SUT, it is important that the load testing tool is able to put the SUT under a steady load with low variance. The behavior of the load during measurements is evaluated in detail in Section 6.2.1.

5.3.5. R

The R project for statistical computing [R] is a software environment for statistical calculations and plotting. It offers a TCP server implementation called Rserve. In this thesis R is used for all kinds of calculations, such as means, median, standard deviation, apriori algorithm and principal component analysis. Since the libraries are well documented and tested, the use of R helps avoiding implementation errors. Even easy calculations such as mean or variance hold a risk of being faultily implemented when looking at metrics such as available filesystem space in bytes over 400 measurements.

5.3.6. Regression injection

Concerning regression injection, this thesis strives to insert common and realistic performance regressions into the microservice system. The regressions were inserted into the carts microservice. The carts microservice was chosen because it was written in Java and is easy to adapt. It is connected to a MongoDB database. The carts microservice was chosen as well for the general research of metric behavior in subsection 5.5.2. The injected regressions were inspired by performance antipatterns as e.g., researched by Smith and Williams [SW03] as well as the work of other performance regression detection research papers [MHH13] [Ngu+12]. The implementations of the antipatterns "The Ramp" and "One-lane-Bridge" were inspired by pseudo-code from a paper of Keck et al. [Kec+16]. Section 5.2 shows a tabular overview of the different kinds of regressions which were inserted.

5.4. Metrics

This section answers research question 1.1 "Which metrics are available and commonly collected?". It describes which metrics are commonly available and explains what the

Table 5.2.: The different kinds of injected regressions

Type of regression	Description
System print [Ngu+12]	Adding unnecessary logs in the system standard output. Comparable to setting a wrong logging level in production.
DB connection [Ngu+12]	A wrong configuration in the database client limits the number of concurrent open connections to 1 opposed to the default of 100.
One-lane bridge [Kec+16]	Occurs when a bottleneck of concurrency exists in the program. To inject this regression semaphores were added at critical points in the program.
Ramp [Kec+16]	Occurs when processing time increases over time. This regression was artificially injected inspired by [Kec+16].
Unnecessary processing [SW03]	A calculation does some heavy processing which would not be necessary. For example evaluating additional data after a searched result already was found.
Increased memory usage [Ngu+12]	The systems memory usage increases significantly. This may be due to a so-called memory leak, which prevents increasing memory loads to be released.

different metrics measure. It offers reasoning for the fact that some metrics are ignored in the performance regression detection set up.

5.4.1. CPU

CPU metrics are collected by use of the Kubernetes [Kubb] default Container Cluster Monitoring tool Heapster [Heab]. In Kubernetes, an important unit concerning CPU usage is millicores (m). Millicores describe a fractional usage of one single core, vCore or hyperthread, depending on the base system. A container which requests 100 m is guaranteed half as much CPU as one asking for 200 m [Kubd]. The available CPU metrics on pod level are listed in Table 5.3 [Heaa].

The metric cpu/usage_rate can be considered to be the microservice equivalent of common CPU usage. The metric cpu/usage is not considered in this work, since its cumulative nature would lead to a need for special handling. Furthermore, its data represents an integral over the cpu/usage_rate.

limit	The limit of available millicores for the pod.
request	The guaranteed amount of available resources meassured in millicores.
usage	The cumulative CPU usage on all cores.
usage_rate	The CPU usage of all cores at a certain point of time measured in millicores.

Table 5.3.: Available CPU metrics in Heapster

Table 5.4.: Available memory metrics in Heapster

limit	The limit of available memory for the pod measured in bytes.
major_page_faults	The cumulative number of major page faults of the pod. A major page fault occurs when memory has to be loaded from the disk.
major_page_faults_rate	The number of major page faults which occurred in the pod in one certain second. A major page fault occurs when memory has to be loaded from the disk.
page_faults	The accumulative number of page faults of the pod. A page fault occurs when memory is already available but has to be mapped by the operating system.
page_faults_rate	The number of major page faults which occurred in the pod in one certain second. A page fault occurs when memory is already available but has to be mapped by the operating system.
request	The guaranteed amount of memory resources to be available measured in bytes.
usage	The total memory usage of the system measured in bytes.
working_set	The working set of the pod measured in bytes. The working set describes all referenced memory of the pod.

5.4.2. Memory

Memory metrics are collected by use of the Kubernetes [Kubb] default Container Cluster Monitoring tool Heapster [Heab]. If feasible, the unit of memory metrics is bytes. The available memory metrics on pod level are listed in Table 5.4 [Heaa].

Table 5.5.: Available filesystem metrics in Heapster

usage	The total number of bytes used on the filesystem.
limit	The total size of the filesystem.
available	The number of remaining bytes in the filesystem.

Table 5.6.: Available network metrics in Heapster

rx	The total number of incoming network bytes.
rx_errors	The total number of errors concerning incoming traffic.
rx_errors_rate	The number of errors concerning incoming traffic per second.
rx_rate	The number of incoming network bytes per second.
tx	The total number of outgoing network bytes.
tx_errors	The total number of errors concerning outgoing traffic.
tx_errors_rate	The number of errors concerning outgoing traffic during on second.
tx_rate	The number of outgoing network bytes per second.

The metrics major_page_faults and page_faults are not considered in this work, since their cumulative nature would lead to a need for special handling. Furthermore, their data is represented in the according rate metrics.

5.4.3. Filesystem

Filesystem metrics are collected by use of the Kubernetes [Kubb] default Container Cluster Monitoring tool Heapster [Heab]. If feasable, the unit of filesystem metrics is bytes. The available filesystem metrics on pod level are listed in Table 5.5 [Heaa].

5.4.4. Network

Network metrics are collected by use of the Kubernetes [Kubb] default Container Cluster Monitoring tool Heapster [Heab]. If feasible, the unit of network metrics is bytes. The available network metrics on pod level are listed in Table 5.6 [Heaa].

Table 5.7.: Collected response metrics in Locust

status_code	The HTTP status code of the response
reason	The Reason-Phrase of the response (OK/Accepted/Not found/...)
url	The full request url of the request.
path_url	The relative url of the request.
method	The method of the request (GET/POST/DELETE).
elapsed	The elapsed time of the request in seconds.

5.4.5. Response time

Response time metrics are collected by use of the load driver Locust[Loc]. To implement this logging, an establishing and writing to the InfluxDB was added to the Python load testing script. The collected metrics are shown in Table 5.7.

5.5. Description of results

The research question of which metrics are commonly collected in a microservice setup was already answered in Section 5.4. Section 5.5.1 describes work on research question RQ 1.2. Section 5.5.2 describes work on research question RQ 1.3.

5.5.1. General metrics behavior

Deviation of measurements during runs

To answer research question RQ1.2 “How stable are metrics during a run?” the distributions of the single relevant measurements were examined. The corresponding plots can be found in Appendix A. Figure 5.3 is shown as an example for those plots. The plot shows the distibution of the measurements divided by its median value. This is done to facilitate recognizing relative deviations. The measurements exclude a 40-minute startup time to avoid showing influences of data which were not in steady state. The red lines show the 25% and 75% quantiles. None of the observed metrics had high deviations of their median values. Only memory/page_fault_rate had some more significant outliers. No deviations were found out of which major issues for performance regression detection were found.

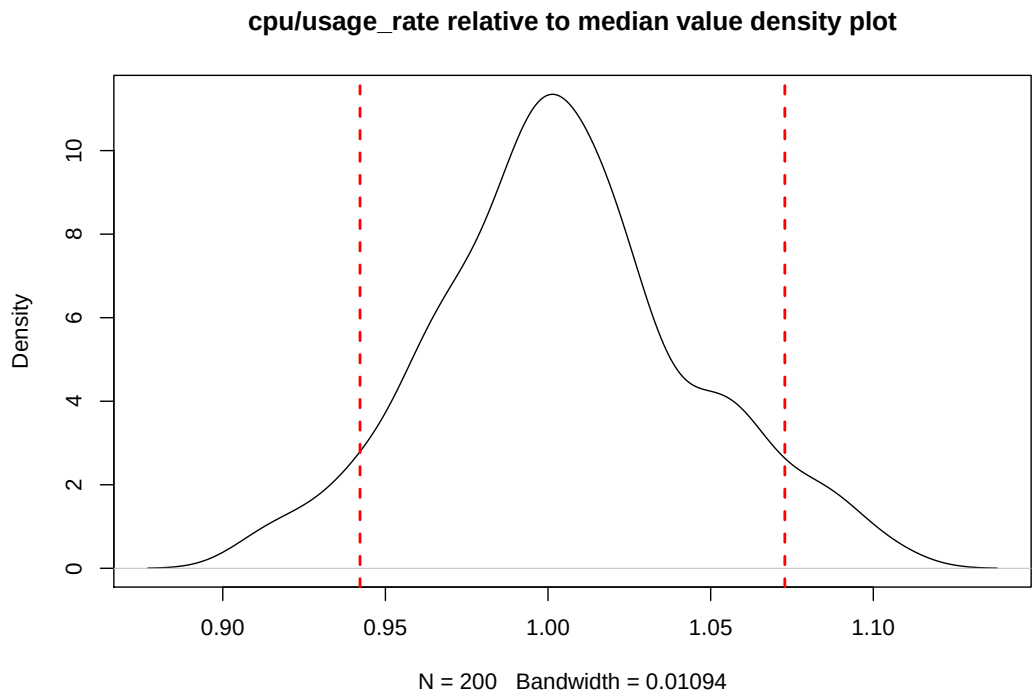

Figure 5.3.: Cpu/usage_rate distribution relative to median value

Normal distributed metrics Nguyen et al. [Ngu+12] mention approaches to filter performance measurements, so that the resulting data is of normal distribution. They do this because their approach is based on control charts which need normal distributed data as an input. In their work, they saw that around 88% of their runs have a bi-modal distribution. 66% of the runs are not of normal distribution concerning the Shapiro-Wilk test ($\alpha = 0.05$). After their filtering approach, they could raise the number of runs which passed the Shapiro-Wilk test to 91%. The main idea for their filtering approach was that they expected the bi-modal distribution to result out of phases where the system idles and performs bookkeeping tasks. Therefore, they propose their filtering technique of removing the data sets which are represented in the first peak of the distribution of the measurements.

To evaluate whether their findings can be reproduced in the microservice environment of this thesis, the measurements of the single runs of the 19 redeployments were evaluated. The findings are depicted in Table 5.8.

The first row shows the number of metric runs. This terminology is chosen to describe one set of measurements for a single metric and run. Out of those metric runs a majority describe metrics which are not relevant because they show no variance. Those are for example metrics such as cpu/request, cpu/limit, memory/limit which are static during

Table 5.8.: Normal distribution findings

number of metric runs	548
runs with variance $\neq 0$	110
runs not normal before filtering	110
runs not normal after filtering	97

runtime or metrics such as filesystem/available for microservices which do not interact with the filesystem. All of those metric runs show a p-value ≤ 0.05 when performing a Shapiro-Wilk test and are therefore not considered to be of a normal distribution. After applying the filtering technique, 97 of those 110 still do not pass the Shapiro-Wilk test. Those findings are mostly expected. The used dataset already removed the startup and shutdown phases of the load tests and the system was put under static load. The idle phases which the filtering tries to eliminate, should already have been removed. Nonetheless, this shows that the majority of collected metrics can not be considered to be of normal distribution. Two of the described approaches, the Student t-test as well as the control chart based approach therefore are at least on a theoretical level not applicable.

5.5.2. Metrics behavior with redeployments

Behavior of CPU metrics

Figure 5.4 shows the behavior of the cpu/usage_rate metric concerning different redeployments of the same system configuration in the carts microservice. The median is built upon the test data with the first 40 minutes removed (5.2.1, A.1). 40 minutes into the test runs there are no trends observable in the cpu/usage_rates any more. The single points show the corresponding median value during one single test run. The red line shows the mean value of the median values of the runs for orientation purposes. Between the maximum and minimum median value of those 19 test runs, lies a range of 123 millicores. Compared to the mean median value of 113 millicores this range is highly relevant. The variance of the data set is 703 and therefore comparably high. The plot suggests that the recorded data has patterns. The median values are strictly alternating between higher and lower values. Furthermore, there are three clusters visible throughout the test runs. Concerning the alternating behavior of the measurements, Section 6.2.2 offers a possible explanation, although the reason for the different clusters would still be unclear.

Figure 5.5 shows the relative deviations of the median CPU measurements throughout the runs compared to the relative deviations of the median requests per minute of the

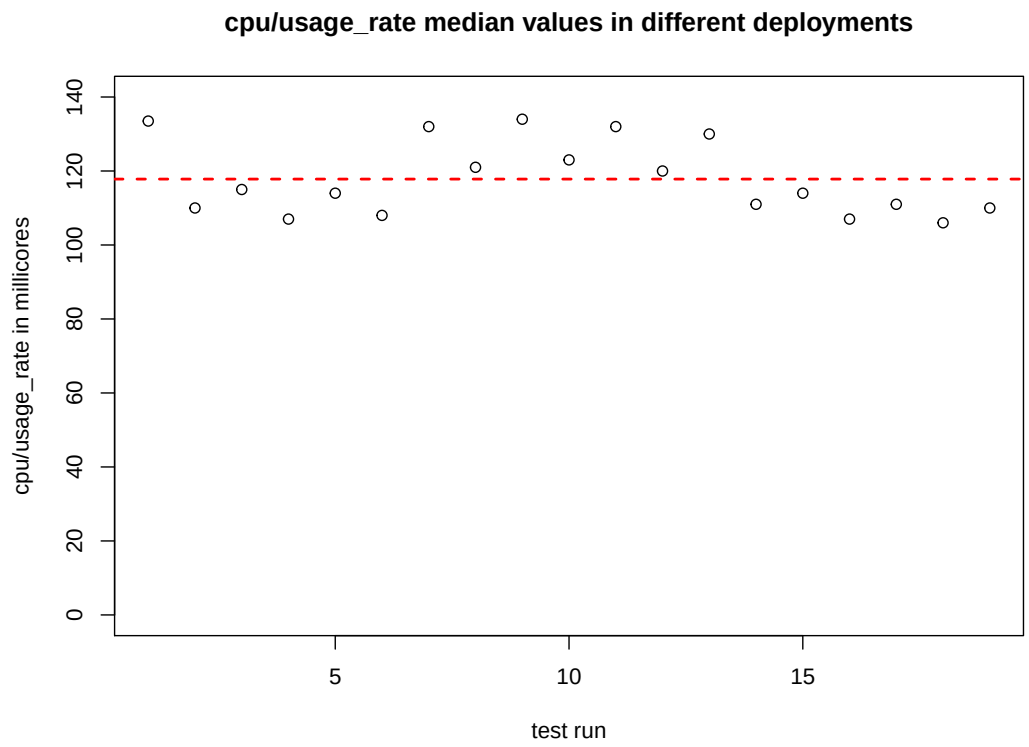

Figure 5.4.: Cpu/usage_rate median in different deployments

load driver. It is clearly visible that the relative deviations of the cpu_usage_rate are significant higher than the relative deviations of the requests per minute of the load driver.

Behavior of memory metrics

Figure 5.6 shows plots of the memory usage of the 19 test runs. Every different line shows the observed behavior of one single test run. This depiction is used because the memory/usage does not reach a clear steady state. As clearly visible, the memory/usage plots all show a trend. Therefore, a visualization of the median values is not able to show a clear picture of the behavior. In the figure, it is clearly visible that memory usage varies enormously between different deployments. Figure 5.7 shows the distributions of absolute relative deviation of the median values of the runs. For orientation purposes, the first violin chart shows the distribution of the median values of the requests per minute in the different deployments. The figure shows significant differences between the distribution of the requests per minute and the memory usage as well as the memory working set. Although the median requests per minute only deviate by a maximum of around 3%, the median measurements of memory usage and working set deviate by a maximum of around 14 % during the recorded test runs. This is a finding which could

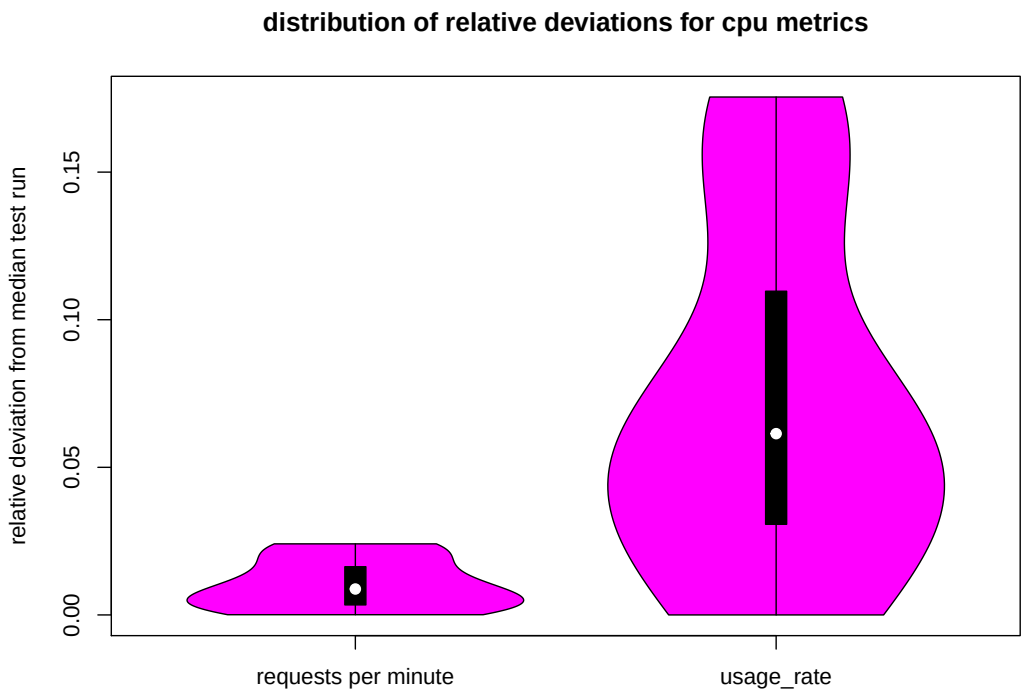

Figure 5.5.: Relative deviations of median CPU measurements during runs compared to requests per minute of the load driver

be a topic for future research.

Behavior of filesystem metrics

None of the observed microservices produced significant measurements concerning filesystem usage. The load testing scripts only worked with a fixed number of articles and used only a fixed number of user accounts. This setting was chosen to reduce the effect of random action selection. Therefore, at this point no conclusions concerning filesystem metrics are available.

Behavior of network metrics

Figure 5.8 shows plots of the network rx and tx rate of the 19 test runs. Every different line shows the observed behavior of one single test run. Opposed to the memory usage and working set, there's no clear relationship in between the different runs visible. Even with different kinds of smoothing the plots do not suggest that the noisy behavior is similar between the different runs. Nonetheless one can see that the measurements for network rx and tx rate do not show trends in the different runs. Figure 5.9 shows

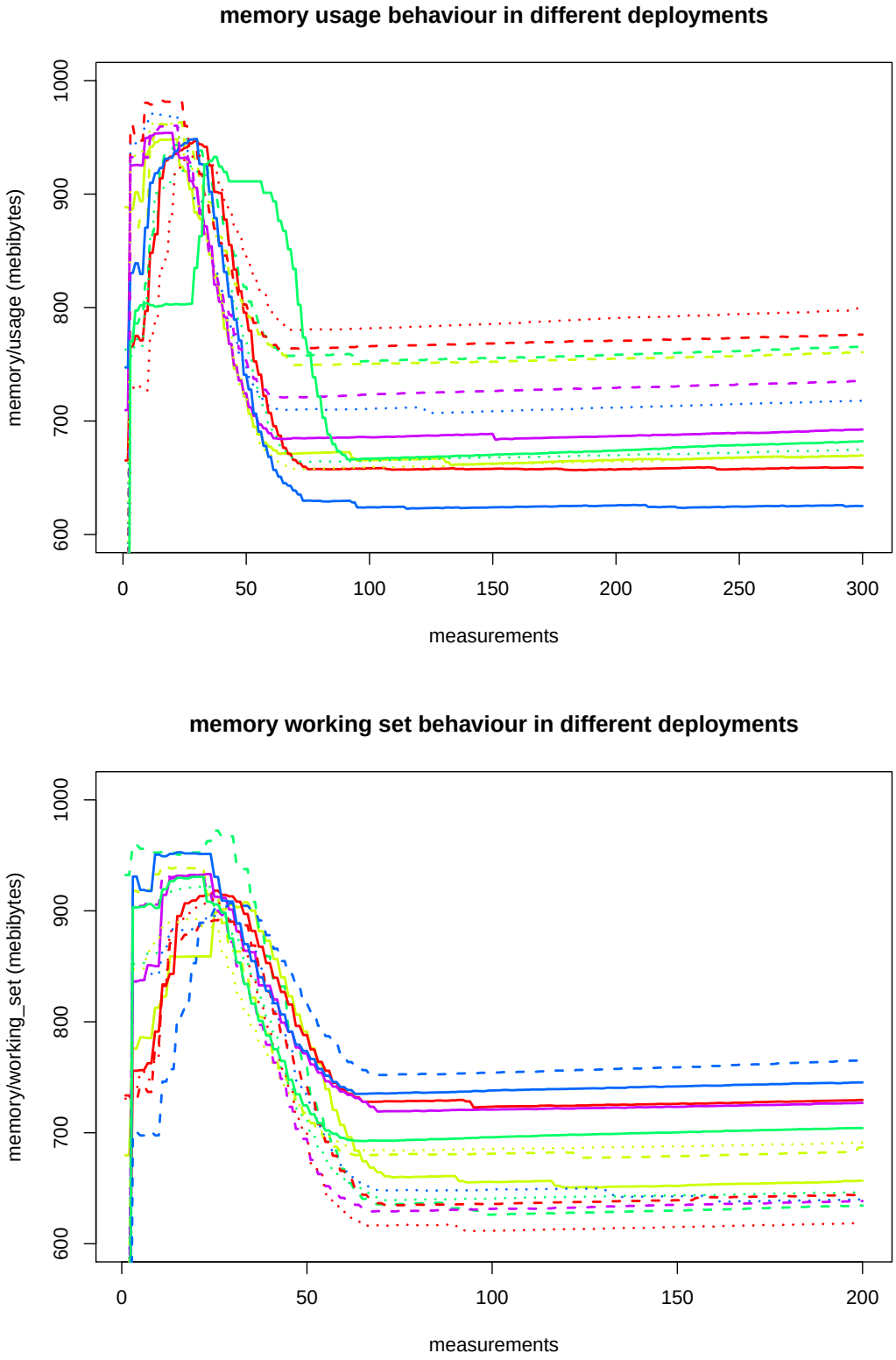

Figure 5.6.: Memory usage and working set behavior during different deployments

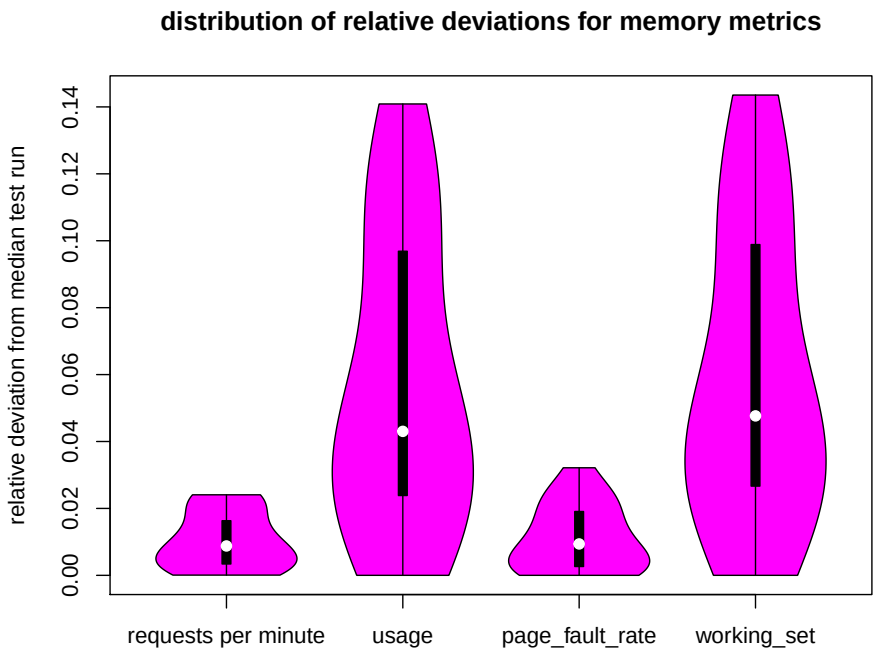

Figure 5.7.: Relative deviations of median memory measurements during runs compared to requests per minute of the load driver

the distributions of absolute relative deviation of the median values of the runs. For orientation purposes, the first violin chart shows the distribution of the median values of the requests per minute in the different deployments. The figure shows that there is no significant difference between the relative deviations between the network metrics and the requests per minute of the load driver. This suggests that the deviations of network metrics may have only been implicated by variance of load during the different runs.

5.5.3. Approaches

In the following tables, the results show in how many of the total 39 test cases the single approaches reported which results. R1 to R6 represent the different types of regressions.

They stand for:

R1: mongo database misconfiguration

R2: unnecessary processing

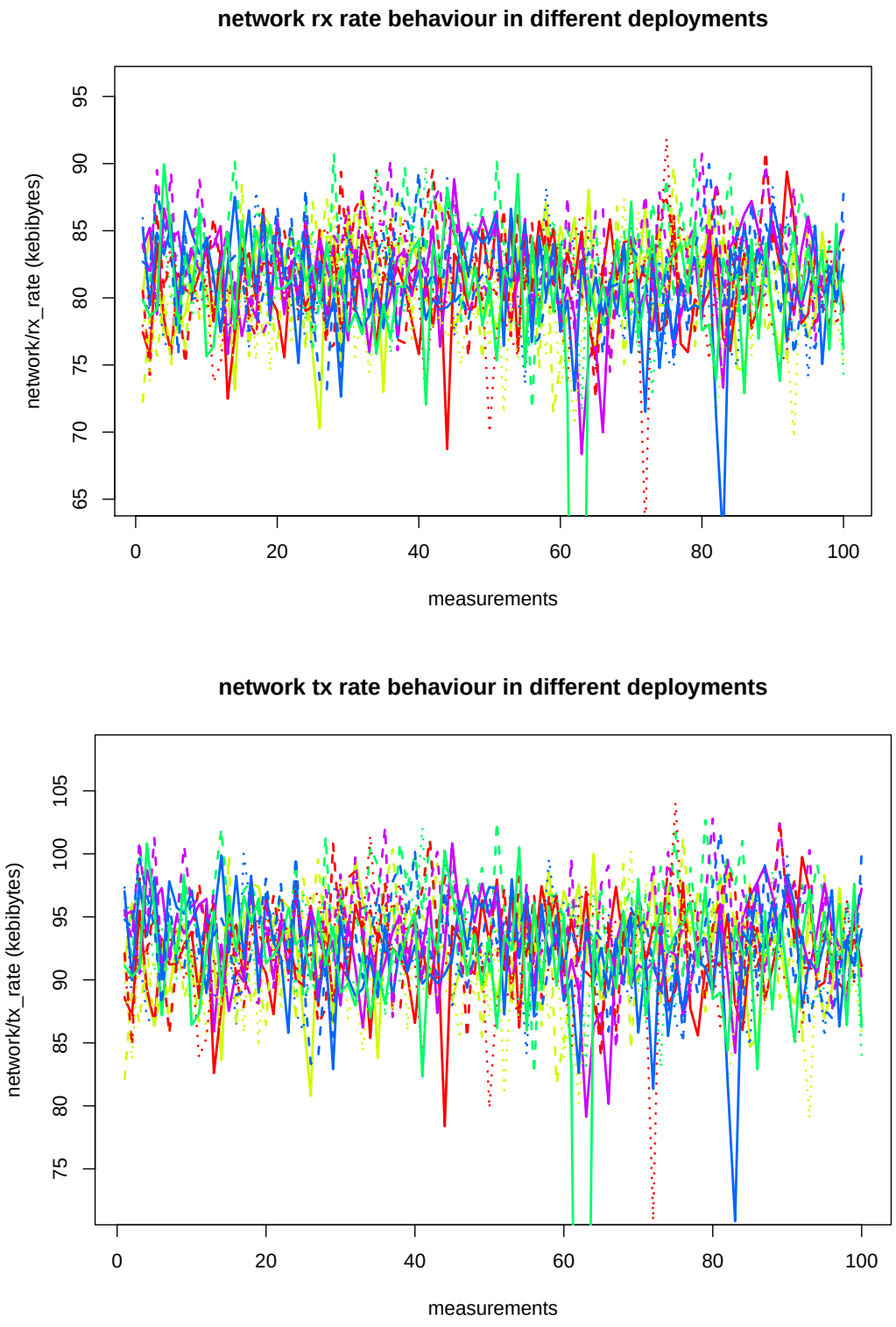

Figure 5.8.: Network rx and tx rate behavior during different deployments

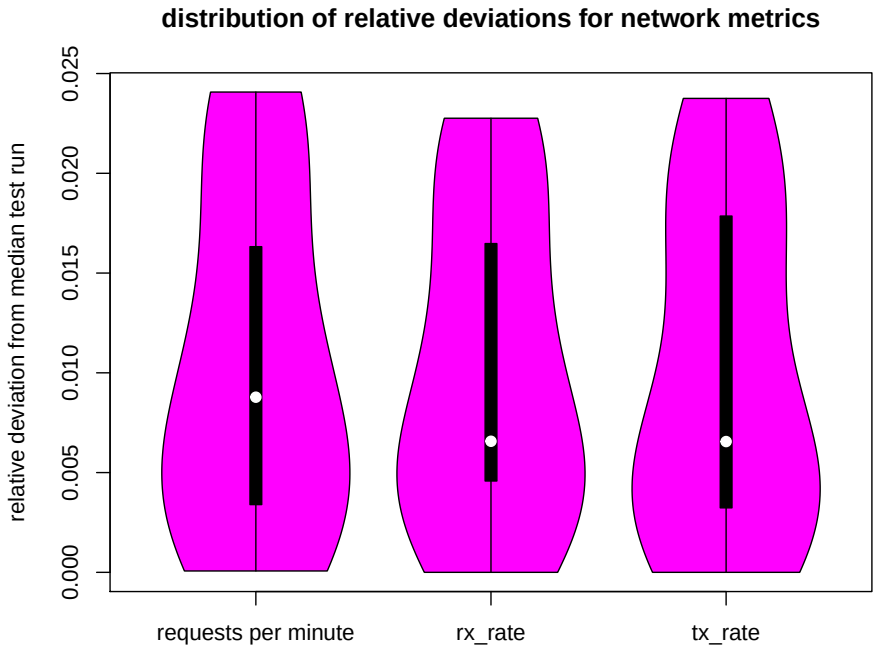

Figure 5.9.: Relative deviations of median network measurements during runs compared to requests per minute of the load driver

R3: a one line bridge antipattern injection

R4: unnecessary memory usage

R5: unnecessary system prints

R6: a the ramp antipattern injection

Since the load testing script was focusing on putting load on the carts microservice, some parts of the system were nearly under no load. Since some approaches struggle with such measurements, measurements with a median cpu/usage_rate below 10 millicores were removed. Furthermore, the network tx_rate and rx_rate triggered a lot of false positives. These metrics were therefore ignored by the approaches. The zipkin microservices were ignored as well because of too low load.

If an approach issued one or more performance regression alerts, a regression was found. If no regression alert was triggered, the approach reported no regression in the system.

Concerning the size of the regressions, R2, R4 and R6 triggered significant changes to some metrics' measurements and were expected to be easy to spot. R1, R3 and R5 did

Table 5.9.: Performance evaluation of the four performance regression detection approaches

Approach	t-test-based	process-controll-based	signature-based	mining repositories-based
TP	24	16	16	7
TN	0	9	8	10
FP	15	6	7	5
FN	0	8	8	17
Precision	63%	73%	70%	58%
Recall	100%	67%	67%	29%
F-measure	77%	70%	68%	39%
Time needed	3 min	3 min	2 min	26 min

Table 5.10.: Performance of Student t-test regression detection

	No change	R1	R2	R3	R4	R5	R6
Regression detected	15	4	4	4	4	4	4
No regression detected	0	0	0	0	0	0	0

not have a too significant influence of the system's behavior those three were expected to be useful for a more fine-grained evaluation of sensitivity of the approaches.

Table 5.9 shows the resulting cumulative true positives, true negatives, false positives, false negatives, precision, recall, and F-measure values of the approaches.

Student t-test based performance regression detection Table 5.10 shows the detection results of the Student t-test-based regression detection. The performance metrics of the approach are depicted in Table 5.9. The Student t-test did not perform well in this evaluation. With an α-value of 0.05 the Student t-test issued a performance regression alert in every set of the test runs. Although the number of detected regressions per test set varied, they resulted in a range of 46 to 84 regression alerts per test set. It therefore is far from reaching the needed zero alerts for detecting no regression.

The evaluation metrics for the Student t-test-based approach are depicted in Table 5.9. The approach has the overall highest F-measure. This is caused by the 100% recall which the approach reached by reporting every possibility as a performance regression.

Statistical process control techniques using machine learning

Table 5.11.: Performance of statistical process control regression detection

	No change	R1	R2	R3	R4	R5	R6
Regression detected	6	2	4	2	2	2	4
No regression detected	9	2	0	2	2	2	0

Table 5.12.: Performance of signature-based performance regression detection

	No change	R1	R2	R3	R4	R5	R6
Regression detected	7	0	1	4	3	4	4
No regression detected	8	4	3	0	1	0	0

The results of the evaluation runs of the statistical process control-based regression detection approach are depicted in Table 5.11. It detected 16 out of 24 of the runs with regressions, although six false positive performance regression alerts were issued as well. Every type of regression was at least detected once. Concerning the different kinds of regressions, it was able to detect every kind of regression. As expected, the more significant regressions R2 and R6 were detected in every of the 4 runs. Although R4, increased memory usage, was considered easy to spot as well, only 2 of the 4 runs were categorized right. Overall, it performed better than the Student t-test.

The evaluation metrics for the Student statistical process control-based approach are depicted in Table 5.9. The approach has with 73% the overall highest observed precision. It reached a recall rate of 67%.

Signature-based performance regression detection

The detection results of the signature-based approach are depicted in Table 5.12. It was one of the fastest regression detection approaches. It did build in between 10 to 13 principal components for signature generation. The evaluation metrics for the signature-based approach are depicted in Table 5.9. The results show a quite similar performance of this approach as the statistical process control-based. Its performance metrics are only faintly lower. These differences are not significant.

One of the biggest reasons of why this approach did not perform better is probably the high and changing number of principal components that had to be to considered to reach a cumulative variance of 90%. The number of used principal components varied between 10 and 13. The order of the principal components is relevant to this approach, but changed in between runs as well. This behavior lead to the fact that only a small number of metrics per principal component did fit to each other. The regression detection therefore was quite limited.

Table 5.13.: Mining performance regression testing repositories performance

	No change	R1	R2	R3	R4	R5	R6
Regression detected	5	0	4	1	0	2	0
No regression detected	10	4	0	3	4	2	4

Mining performance regression testing repositories

The detection results of the mining performance regression testing repositories approach are shown in Table 5.13. Its evaluation metrics are depicted in Table 5.9. The approach of mining performance regression testing repositories did not perform well. Its precision of 58%, recall of 29%, and F-measure of 39% were the worst of all researched approaches. Three out of six types of regressions were not detected at all (R1, R4, and R6). Two of those three, R4 and R6, were in the categories of high significance and should have been easy to spot. It performed worse than all other approaches. It was not able to detect three out of 6 types of regressions at all. Although it did have the highest number of true negatives in this evaluation, it seems that it simply had a bias to reporting no regression. Additionally, this approach was significantly slower. The reason for that is the used apriori algorithm, which does not scale well to an increasing number of input elements. For an isolated evaluation of a single microservice's metrics, the time which the approach needed was not significantly longer than to other approaches. In such a setup, all approaches needed around 2 minutes. The approach did need a lot of RAM (4 GB) and time (26 min). The other approaches needed 3 or less minutes and 2 GB of RAM each.

5.6. Discussion of results

5.6.1. General metrics behavior

The findings suggest that there is no unexpected behavior of performance metrics during a run. The measurements did not deviate significantly. Therefore the existent deviations should not be challenging to the performance regression techniques.

5.6.2. Metrics behavior with redeployments

Results for CPU metrics

The observed behavior of the cpu/usage_rate concerning redeployments has huge implications for performing performance regression detection in a microservice setups based on cpu/usage_rate. The difference in metric results between redeployments is high enough to trigger performance regression detection approaches, although the system was not changed at all. Furthermore, a CPU performance regression may be hidden by the fact, that the cpu/usage_rate strives for a lower level in this certain testing deployment. A possible but very time-expensive solution to this problem would be to run several test runs throughout several redeployments. This solution would stand opposed to the very frequent changes, which can be observed in microservice systems.

Further research should be done on why such behavior can be observed. What does influence the metrics? Are different devisions between the nodes of the cluster and the microservices a reason for the high variance? Does the busy cluster lead to such high noise in the recorded median values? Which alternative CPU metrics could be collected to reach more stable measurements between redeployments?

Results for memory metrics

The observed behavior of the memory/usage has huge implications for performing performance regression detection. The measurements of the different runs strive for different target values. This leads to similar issues as with the cpu/usage_rate.

Results for network metrics

The observed behavior of the network metrics does not have huge implications for performing performance regression detection. The measurements of the different runs vary approximately equally as the produced load of the load driver.

5.6.3. Approaches

Student t-test based performance regression detection

These measurements are a good example of why the F-measure can not blindly be trusted for evaluation. In the set of test cases the distribution of tests with a regression and without a regression is approximately equal. In reality, one would expect that the big majority of changes to a software system do not introduce performance regressions to the system. In such a more realistic setup the precision of the approach would drop significantly. In the current setup, the precision of the approach is already the second

worst, this would be even worse by using a more realistic ratio of tests without and with regressions. The approach probably reports that many performance regressions because its foundation needs normal distributed data. As this work showed, the performance measurements of the microservice environment were not normal distributed. Furthermore, the deviations of metrics resulting out of redeployments probably had a negative influence as well. The approach of Student t-test performance regression detection in microservice systems, is considered to be not applicable at all. A performance regression detection approach with such a high number of false positives would be of no value in a realistic setup.

Statistical process control techniques using machine learning

Although the precision of 73% is comparable to the findings of the original work [Ngu+12], the recall of 67% is quite low for a practical application. These findings are to some amount surprising, since it was observed that the input data was not of normal distribution. Normal distributed data sets are a requirement for control charts, which the approach is based on. During development it was observed, that the sensitivity of the approach highly depends on the size of the load testing repository. Small repositories lead to a high number of false positives. It is suspected that one of the main reasons why this control chart-based approach was outperforming the Student t-test that significantly, is that the control chart-based approach was able to make use of the whole load testing repository, while the Student t-test only used one test out of the repository as reference. Since this approach is easy to understand and had some promising results, it may be a possible solution for performance regression detection in microservice environments. Nonetheless, the observed performance is considered to be too low for practical applications.

Signature-based performance regression detection

The approach seemed to be very fast and may therefore be promising for future research. Nonetheless a solution for the failing mapping of old and new principal components would have to be found. One huge disadvantage of the approach is that the weights of metrics in a principal component are a concept not as easily understood as for example violation ratios. Therefore, it is sometimes not easy to understand why a performance regression was reported.

Mining performance regression testing repositories

It is suspected that the bad performance concerning true positives, is as well based on the big data set. The more different measurements have to be consider when extracting the association rules, the more rules may be lost compared to a smaller subset of such measurements. The thresholds for selection rules are support meaning the relative

proportion of the dataset where the rule occurs and confidence describing the probability that the conclusion of the rule holds true. Support may be sensitive to increasing data set sizes. For smaller test evaluations, the observed performance of the approach was better. Nonetheless because of its inability to scale, it does not make sense to apply the approach to a setting like microservice environments.

Chapter 6

Threats to validity

This chapter gives an overview over the different threats to validity of this work. To every threat an explanation is provided, what was done to reduce the impact of the threat.

6.1. External validity

External validity describes how far the findings of a work can be generalized. The remainder of this section will focus on such possible threats [Woh+12].

6.1.1. Sock shop environment

The Sock Shop System [Soc] is as Aderaldo et al. [Ade+17] mention in their work, not a perfect system for microservice. They conclude that no system is currently major enough to be used as a community wide benchmark. Some of the conclusions of this thesis may be influenced by the chosen system under test and describe special properties of the Sock Shop system. Distributions and measurements may be unique to this system. Since Sock Shop was implemented as a microservice demonstration platform the associated risk is considered to be reasonably low.

6.1.2. Kubernetes and Docker

This work focuses on the container orchestration tool Kubernetes and its used default monitoring tool Heapster. Kubernetes is built upon the virualization of Docker. Some findings of this work depend on the implementation of Kubernetes (Section

refloadOnCluster). Furthermore, the metrics which are available and collected depend on the used virtualization and the used monitoring tool as well. The findings of this work can not easily be generalized for use in other orchestration, monitoring or virtualization tools.

6.1.3. Regression injection

The injected performance regressions of this thesis are of artificial nature. To evaluate the performance of a single approach, this may lead to bias concerning constructing artificial regressions, in which selected approaches work especially well. Since this thesis focuses on an empirical comparison between approaches, there is no approach which is preferred to work especially well by the conductors. Furthermore, the resulting values are not used to benchmark the approaches independently. The resulting information is only used as an ordinal measure, comparing how well the single approaches behave in direct comparison. These points relativize the influence of the artificial nature of the performance regressions. In terms of generalization, it is unclear how far the findings of this work can be used for detecting real regressions in a system.

6.1.4. Approach reimplementations

The selected approaches had to be reimplemented, because no reference implementations were available. The implementation is highly depended on the referenced research papers. Inaccuracies in the description of approaches, or misinterpretation may have lead to an implementation, which does not behave like the original authors intended. Differences in observations may be based on this fact. The quality of description of the approaches was considered in the selection of approaches. The approaches which were implemented, seemed to be clear in their description in most parts. Some minor parts which were not described in detail, were either not relevant or implemented based on an educated guess.

6.2. Internal validity

Internal validity focuses on the question of to which extend the conclusions of a work are valid. It describes whether the performed actions really caused the observed effect [Woh+12].

6.2.1. Artificial load

Often when performing load tests, one issue considering the validity of results, is whether the system was put under a steady load. Otherwise, the reason for observed deviations could lie in variances of the input load. Sometimes, the test drivers, which simulate users putting load on the system, are not able to put the system under the requested load. Variance in inputs load lead to a variance in measured metrics. This is of special relevance to the observations in Section 5.5.2

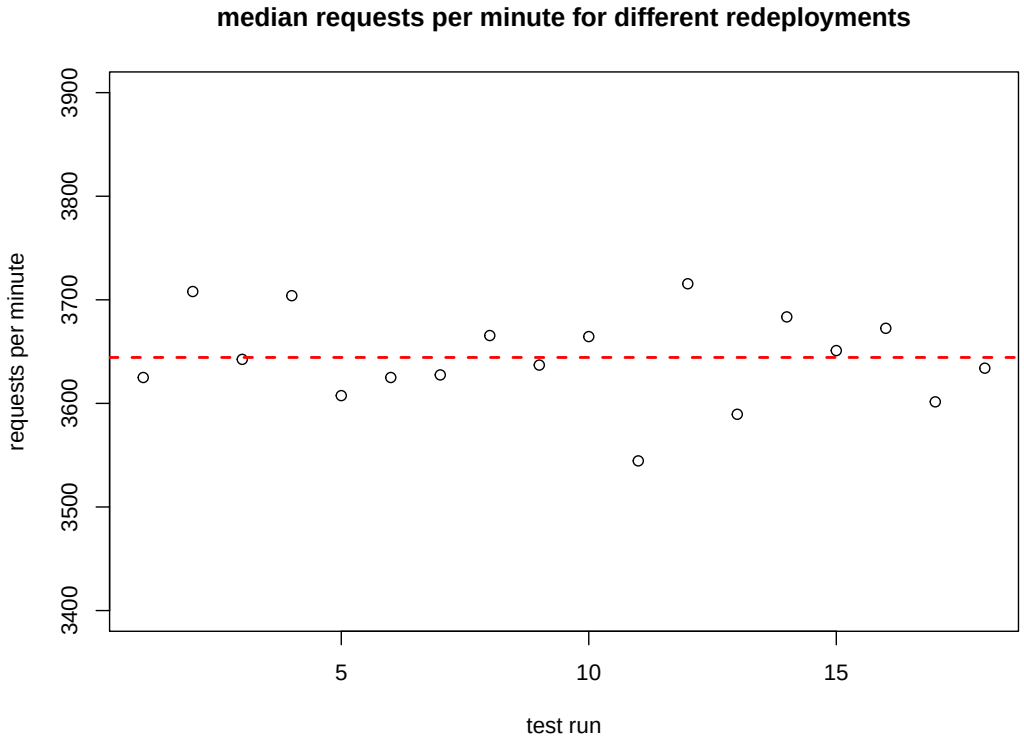

Figure 6.1.: Requests per minute median in different deployments

Figure 6.1 shows the median requests per minute of the Locust load driver. In all runs, the requests per minute reached a nearly constant level after a short warm up time. A comparison between median values therefore seems sensible. The standard deviation of the data set is 48 which is very low in comparison to the mean, which is 3653. Therefore, the deviations of the input load can be considered to be low. Correlations between the patterns, which were observed in Figure 5.4, are not obvious. Although not perfectly constant the median values are distributed in the small range of 172 requests per minute. Therefore, the load is considered to be of sufficient stability between redeployments.

6.2.2. Load on cluster

The cluster itself should not be put under a too high load. If the cluster is not able to cater to the resource requests of the SUT, measurements may be influenced by this fact. Figure 6.2 shows the CPU usage of the three nodes of the Kubernetes cluster throughout the load testing concerning the metric research. The x axis shows the sequential measurements while the y axis shows the relative CPU usage of the whole system. There are several things observable. First of all, the different redeployments can be seen. At the start of each redeployment, all three nodes show a high CPU peak. Between the redeployments, the nodes all reach a CPU usage level which is not critically high. The Kubernetes master shows the highest stationary CPU usage with approximately 60% usage rate during a load test. The second plot shows an excerpt of the nodes' CPU usage rates. A clearly alternating behavior is visible. When the third node (green) has comparably high CPU usage, the first node has a low usage rate. The same holds for high CPU usage of the third and low CPU usage of the first node. This probably depends on the distribution of the single microservices on the different nodes. This behavior may give at least to some degree an explanation to the behavior observed in Figure 5.4. The scheduler of Kubernetes, which decides on where to deploy the single microservices, uses several rules like spreading pods of the same replication controller over several nodes and keeping resource utilization on the different nodes balanced [Kubc]. Beginning with version 1.4 of Kubernetes, some controlling exists, which allows to manually influence how pods are distributed [Kuba]. It may be necessary to restrict this flexibility of Kubernetes to reach more consistent results.

6.2.3. Busy cluster

The load tests of this thesis were performed on an OpenStack cluster of the University of Stuttgart. This cluster is being used for other purposes as well. The test environment therefore was based on a so called busy cluster. Although on the level of virtualization, the system was under no other load, measurements may be influenced by that fact. This may be considered an advantage as well, since realistic environments, can be considered busy as well.

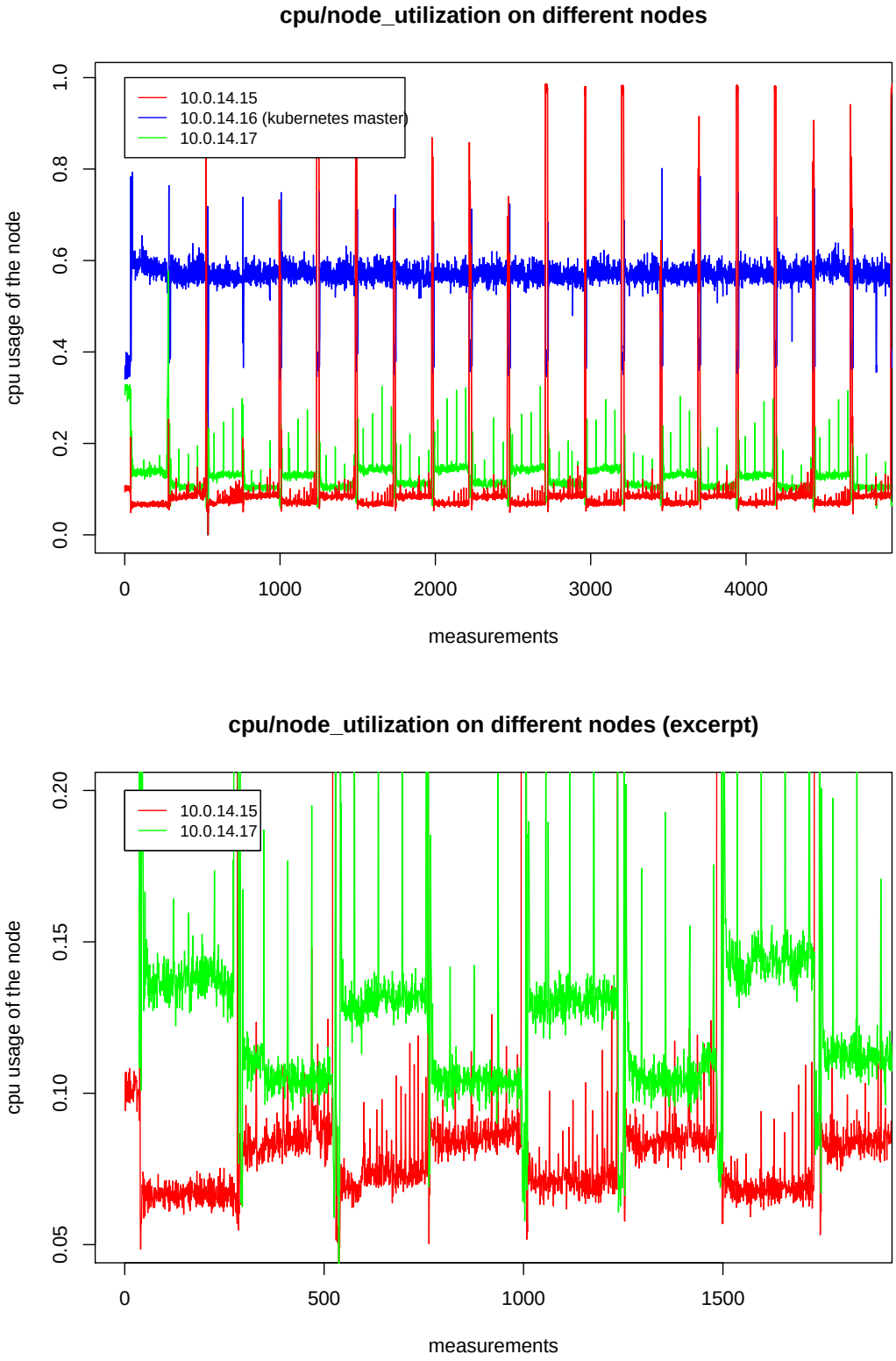

Figure 6.2.: Node CPU usage throughout series of load tests

Chapter 7

Conclusion

7.1. Summary

After giving an introduction to the field performance regression detection and to the field of microservices, this work has given an in-depth overview over research work on approaches to performance regression detection. The different approaches were compared to each other and a subset of them were implemented for a future evaluation. Basic research on the behavior of software performance metrics in between redeployments of a microservice system has been performed. The measurements showed that some key metrics such as CPU usage rate and memory usage show significant deviations resulting out of redeployments.

The evaluation of the different performance regression detection approaches in the microservice environment showed that some existing approaches do not perform well considering recall and precision of the regression detection. Although some approaches such as performance signature-based regression detection and control charts-based regression performed better than others, the results render no approach fit for practical use. In the evaluation, the most promising approach was able to reach a recall of 67% and a precision of 73%.

7.2. Discussion

The most interesting finding of this thesis are the so far unknown performance metric deviations resulting out of redeployments of a microservice system. Although redeployments of the whole system are not commonly performed in a microservice environment,

these findings show that microservice performance measurements have additional parameters influencing them, compared to measurements in a monolithic application.

This work concludes that the deviations resulting out of decisions of the Kubernetes scheduler, which decides how to partition the different microservices on the nodes of the cluster, highly influences the performance of performance regression detection. If this conclusion is true, microservice performance measurements of one microservice may be influenced by other microservices with no connection on a software level, but just are deployed to the same node.

Microservice environments promise some chances to software performance engineers such as isolated observation of performance metrics in a single microservice. But there are some open and even unknown challenges like these probably Kubernetes scheduler-based deviations of measurements.

A more general understanding of microservice performance and the behavior of microservice performance metrics, would help building efficient performance regression detection approaches for microservices.

This work showed that performance regression detection has a need for special solutions in the field of microservice environments. In the following section, some possible future work is presented.

7.3. Future work

Future work could put a focus on how and to which amount the performance measurements of single microservice instances can be influenced by microservices with high load on the same node. A possible solution for avoiding the high deviations in measurements of redeployments, would be to perform the regression detection by only redeploying the changed microservices. It is unclear whether and to which amount past developments in the microservice environment would influence such measurements. Another approach would be to focus on the performance evaluation of single dedicated microservice instances. Challenges in such research would be how to avoid having to build extensive stubs for such an isolated evaluation or how to minimize the influence of other microservices when evaluating in a real microservice environment. In this work, possible approaches to measuring scalability, elasticity, and resilience were presented. Future work could evaluate how well those measurements work in a microservice environment.

Chapter 8

Acknowledgements

This work would not have been possible without André van Hoorn and Teerat Pitakrat of the Reliable Software Systems Research Group at the University of Stuttgart. Especially without the technical support of Teerat Pitakrat setting up the microservice environment would have been a lot more work. Additionally, I have to make a shout out to Cor-Paul Bezemer, who gave valuable feedback and inspired the research on microservice performance metrics behavior. Last but not least, I want to express my thanks to Alberto Avritzer who helped polishing this work with his feedback.

Chapter 8

Bibliography

[Ade+17] C. M. Aderaldo, N. C. Mendonça, C. Pahl, P. Jamshidi. "Benchmark requirements for microservices architecture research." In: *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. IEEE Press. 2017, pp. 8–13 (cit. on pp. 50, 71).

[Ama+15] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, M. Steinder. "Performance Evaluation of Microservices Architectures Using Containers." In: *Proceedings of the 2015 IEEE 14th International Symposium on Network Computing and Applications*. 2015, pp. 27–34 (cit. on pp. 1, 10, 11).

[Bez+14] C Bezemer, E. Milon, A. Zaidman, J. Pouwelse. "Detecting and analyzing I/O performance regressions." In: *Journal of Software: Evolution and Process* 26.12 (2014), pp. 1193–1212 (cit. on p. 29).

[BHJ16] A. Balalaie, A. Heydarnoori, P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture." In: *IEEE Software* 33.3 (2016), pp. 42–52. ISSN: 0740-7459 (cit. on p. 10).

[BPG15] C. P. Bezemer, J. Pouwelse, B. Gregg. "Understanding software performance regressions using differential flame graphs." In: *Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 535–539 (cit. on pp. 29, 35).

[Cam+16] A. de Camargo, I. Salvadori, R. d. S. Mello, F. Siqueira. "An Architecture to Automate Performance Tests on Microservices." In: *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*. iiWAS '16. New York, NY, USA: ACM, 2016, pp. 422–429. ISBN: 978-1-4503-4807-2 (cit. on p. 18).

[CBK09] V. Chandola, A. Banerjee, V. Kumar. "Anomaly detection: A survey." In: *ACM computing surveys (CSUR)* 41.3 (2009), p. 15 (cit. on p. 9).

[CG09] L. Crispin, J. Gregory. *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009, pp. 276 –279 (cit. on p. 6).

[Dbl] *dblp: computer science bibliography*. http://dblp.uni-trier.de/ (cit. on p. 17).

[Dev] *Performance Awareness in Software Development - Research @ D3S - Department of Distributed and Dependable Systems*. http://d3s.mff.cuni.cz/research/development_awareness/ (cit. on p. 28).

[DG06] J. Davis, M. Goadrich. "The Relationship Between Precision-Recall and ROC Curves." In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. ACM, 2006, pp. 233–240. ISBN: 1-59593-383-2 (cit. on p. 8).

[Dül17] T. F. Düllmann. "Performance anomaly detection in microservice architectures under continuous change." MA thesis. 2017 (cit. on p. 17).

[Foo+10] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, P. Flora. "Mining Performance Regression Testing Repositories for Automated Performance Analysis." In: *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*. 2010, pp. 32–41 (cit. on pp. 30, 38, 48).

[Gha+13] S. Ghaith, M. Wang, P. Perry, J. Murphy. "Automatic, load-independent detection of performance regressions by transaction profiles." In: *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*. ACM. 2013, pp. 59–64 (cit. on p. 23).

[Gha+16] S. Ghaith, M. Wang, P. Perry, Z. M. Jiang, P. O'Sullivan, J. Murphy. "Anomaly detection in performance regression testing by transaction profile estimation." In: *Software Testing, Verification and Reliability* 26.1 (2016), pp. 4–39 (cit. on pp. 23, 24).

[GIM17] M. Gribaudo, M. Iacono, D. Manini. "Performance evaluation of massively distributed microservices based applications." In: *Proceedings of the 31st European Conference on Modelling and Simulation, ECMS 2017*. European Council for Modelling and Simulation. 2017, pp. 598–604 (cit. on p. 18).

[HBRM16] S. Hosseini, K. Barker, J. E. Ramirez-Marquez. "A review of definitions and measures of system resilience." In: *Reliability Engineering & System Safety* 145 (2016), pp. 47–61 (cit. on pp. 11, 12).

[Heaa] *heapster/storage-schema.md at master kubernetes/heapster*. https://github.com/kubernetes/heapster/blob/master/docs/storage-schema.md (cit. on pp. 53–55).

[Heab] *kubernetes/heapster: Compute Resource Usage Analysis and Monitoring of Container Clusters*. https://github.com/kubernetes/heapster (cit. on pp. 51, 53–55).

[Hei+17] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger. "Performance Engineering for Microservices: Research Challenges and Directions." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. New York, NY, USA: ACM, 2017, pp. 223–226. ISBN: 978-1-4503-4899-7 (cit. on pp. 1, 11).

[HKR13] N. R. Herbst, S. Kounev, R. H. Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not." In: *Proceedings of the ICAC*. 2013, pp. 23–27 (cit. on p. 14).

[Hol13] E. Hollnagel. *Resilience engineering in practice: A guidebook*. Ashgate Publishing, Ltd., 2013 (cit. on p. 14).

[Hor+13] V. Horký, F. Haas, J. Kotrč, M. Lacina, P. Tůma. "Performance Regression Unit Testing: A Case Study." In: *Computer Performance Engineering: 10th European Workshop, EPEW 2013, Venice, Italy, September 16-17, 2013. Proceedings*. Ed. by M. S. Balsamo, W. J. Knottenbelt, A. Marin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 149–163. ISBN: 978-3-642-40725-3 (cit. on p. 26).

[Inf] *influxdata/influxdb: Scalable datastore for metrics, events, and real-time analytics*. https://github.com/influxdata/influxdb (cit. on p. 51).

[Isl+12] S. Islam, K. Lee, A. Fekete, A. Liu. "How a consumer can measure elasticity for cloud platforms." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, pp. 85–96 (cit. on p. 14).

[Kec+16] P. Keck, A. Van Hoorn, D. Okanović, T. Pitakrat, T. F. Düllmann. "Antipattern-based problem injection for assessing performance and reliability evaluation techniques." In: *Proceedings ot the Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 64–70 (cit. on pp. 52, 53).

[Kuba] *Assigning Pods to Nodes - Kubernetes* (cit. on p. 74).

[Kubb] *Kubernetes - Production-Grade Container Orchestration*. http://kubernetes.io (cit. on pp. 51, 53–55).

[Kubc] *Kubernetes: Advanced Scheduling in Kubernetes* (cit. on p. 74).

[Kubd] *Managing Computer Resources for Containers | Kubernetes*. https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/ (cit. on pp. 11, 53).

[LEB15] S. Lehrig, H. Eikerling, S. Becker. “Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics.” In: *Proceedings of the 2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. 2015, pp. 83–92 (cit. on pp. 10, 12, 14).

[LL13] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013 (cit. on p. 6).

[Loc] *Locust - A modern load testing framework*. http://locust.io/ (cit. on pp. 51, 52, 56).

[Mal10] H. Malik. “A methodology to support load test analysis.” In: *Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2. 2010, pp. 421–424 (cit. on pp. 21, 37, 38).

[Mei+07] J Meier, C. Farre, P. Bansode, S. Barber, D. Rea. *Performance testing guidance for web applications: patterns & practices*. Microsoft press, 2007 (cit. on p. 5).

[MHH13] H. Malik, H. Hemmati, A. E. Hassan. “Automatic detection of performance deviations in the load testing of Large Scale Systems.” In: *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 1012–1021 (cit. on pp. 21, 52).

[Ngu+12] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, P. Flora. “Automated detection of performance regressions using statistical process control techniques.” In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, pp. 299–310 (cit. on pp. 6, 12, 24, 38, 52, 53, 57, 69).

[Ope] *Home » OpenStack Open Source Cloud Computing Software* (cit. on p. 50).

[Pas+15] A. Pasquini, M. Ragosta, I. A. Herrera, A. Vennesland. “Towards a Measure of Resilience.” In: *Proceedings of the 5th International Conference on Application and Theory of Automation in Command and Control Systems*. ATACCS ’15. New York, NY, USA: ACM, 2015, pp. 121–128. ISBN: 978-1-4503-3562-1 (cit. on p. 14).

[R] *R: The R Project for Statistical Computing*. https://www.r-project.org/ (cit. on p. 52).

[Sha+15] W. Shang, A. E. Hassan, M. N. Nasser, P. Flora. “Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters.” In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. 2015, pp. 15–26 (cit. on pp. 6, 12, 18, 19, 21, 22, 35, 38).

[Soc] *Microservices Demo: Sock Shop*. https://microservices-demo.github.io/ (cit. on pp. 51, 71).

[SS06] R. H. Shumway, D. S. Stoffer. *Time series analysis and its applications: with R examples*. Springer Science & Business Media, 2006, pp. 23–29 (cit. on p. 48).

[Str12] L. Strigini. "Fault Tolerance and Resilience: Meanings, Measures and Assessment." In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by K. Wolter, A. Avritzer, M. Vieira, A. van Moorsel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–24. ISBN: 978-3-642-29032-9 (cit. on pp. 14, 15).

[SW03] C. U. Smith, L. G. Williams. "More new software performance antipatterns: Even more ways to shoot yourself in the foot." In: *Proceedings of the Computer Measurement Group Conference*. 2003, pp. 717–725 (cit. on pp. 52, 53).

[THS11] W. T. Tsai, Y. Huang, Q. Shao. "Testing the scalability of SaaS applications." In: *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 2011, pp. 1–4 (cit. on p. 12).

[Wel47] B. L. Welch. "The generalization ofstudent's' problem when several different population variances are involved." In: *Biometrika* 34.1/2 (1947), pp. 28–35 (cit. on p. 28).

[Wen17a] N. Wenzler. *Automated Performance Regression Detection in Microservice Architectures - code base*. Sept. 2017. DOI: 10.5281/zenodo.890936. URL: https://doi.org/10.5281/zenodo.890936 (cit. on p. 3).

[Wen17b] N. Wenzler. *Automated Performance Regression Detection in Microservice Architectures - Raw Measurements*. Sept. 2017. DOI: 10.5281/zenodo.888699. URL: https://doi.org/10.5281/zenodo.888699 (cit. on p. 3).

[Woh+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012, pp. 102 –110 (cit. on pp. 71, 72).

All links were last followed on September 14, 2017.

Appendix A

Metric measurements

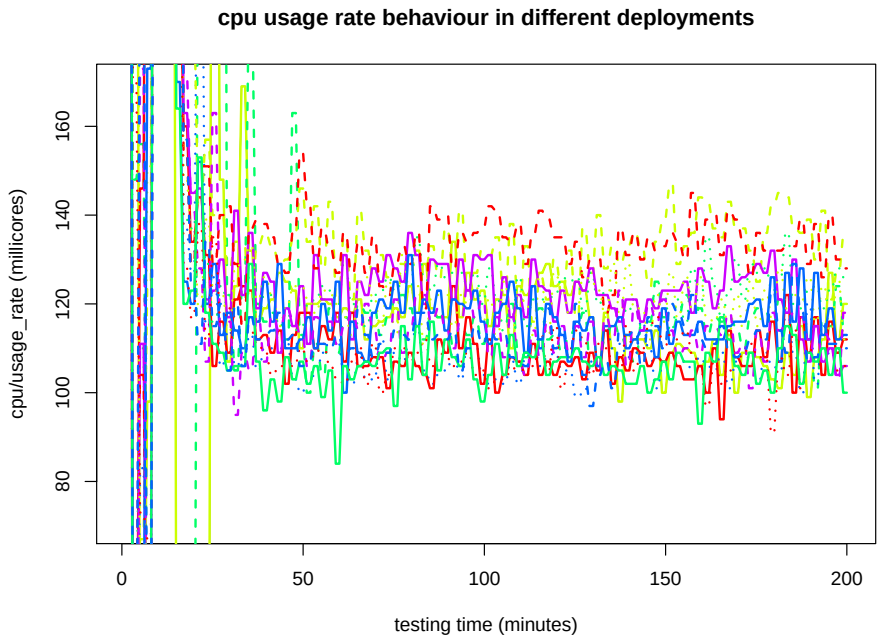

Figure A.1.: Cpu/usage_rate development in different deployments

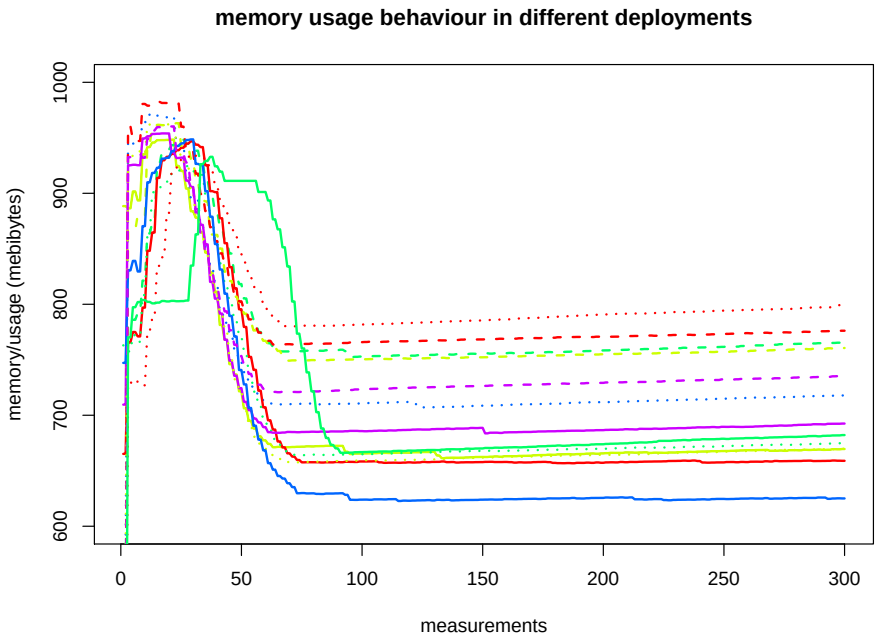

Figure A.2.: Memory/usage development in different deployments

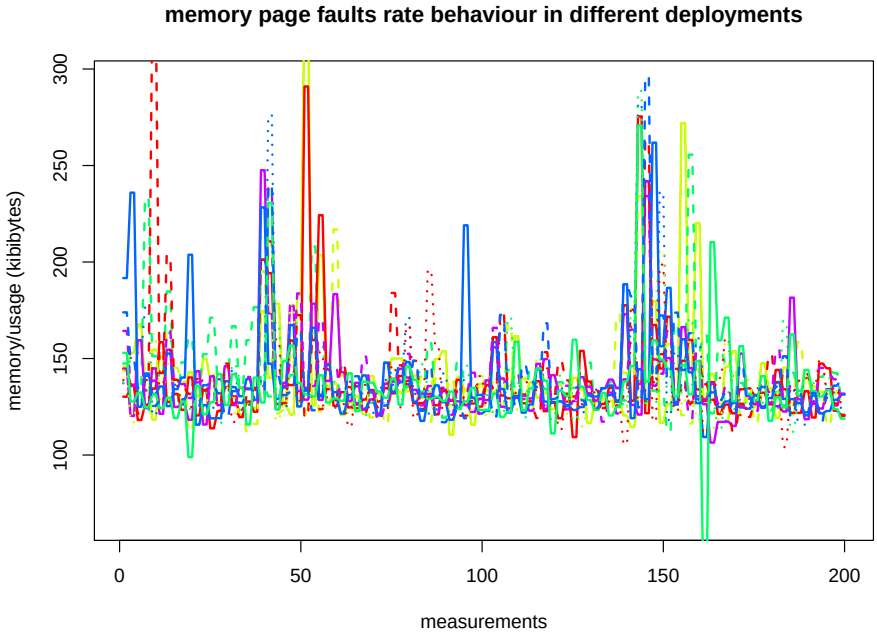

Figure A.3.: Memory/page_faults_rate development in different deployments

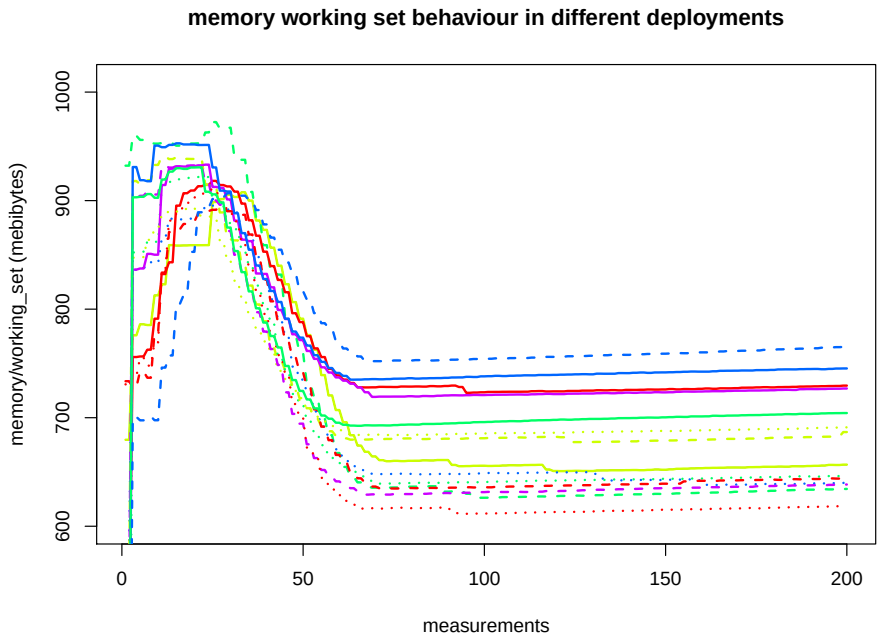

Figure A.4.: Memory/working_set development in different deployments

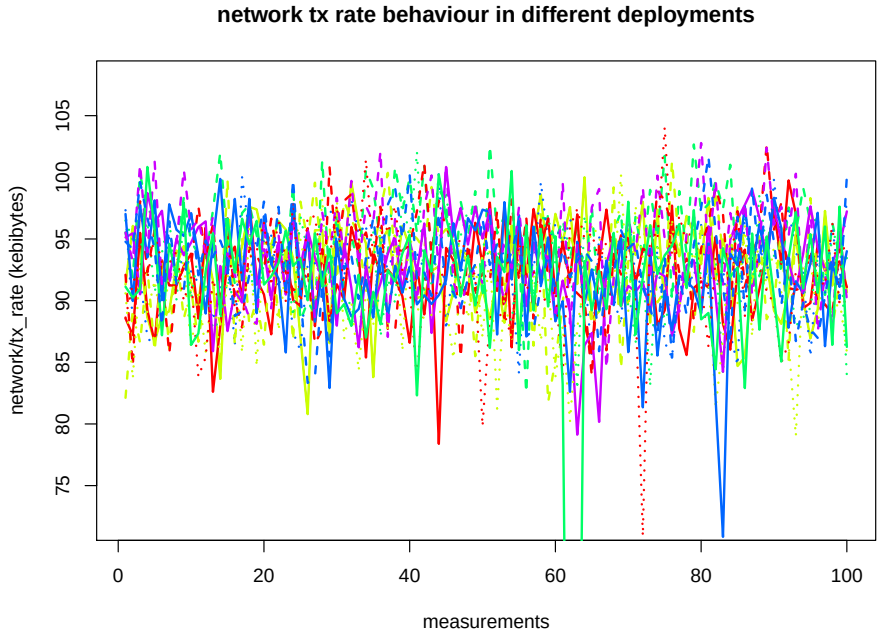

Figure A.5.: Network/tx_rate development in different deployments

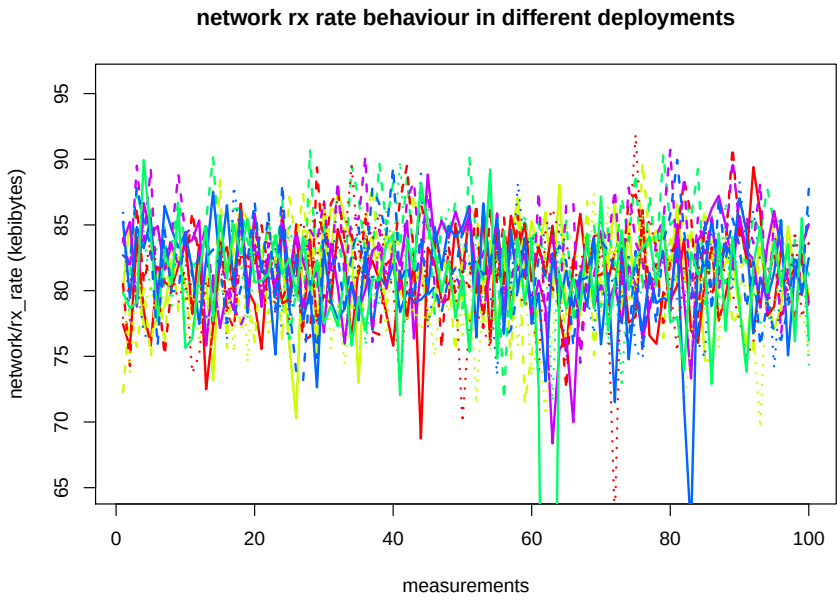

Figure A.6.: Network/rx_rate development in different deployments

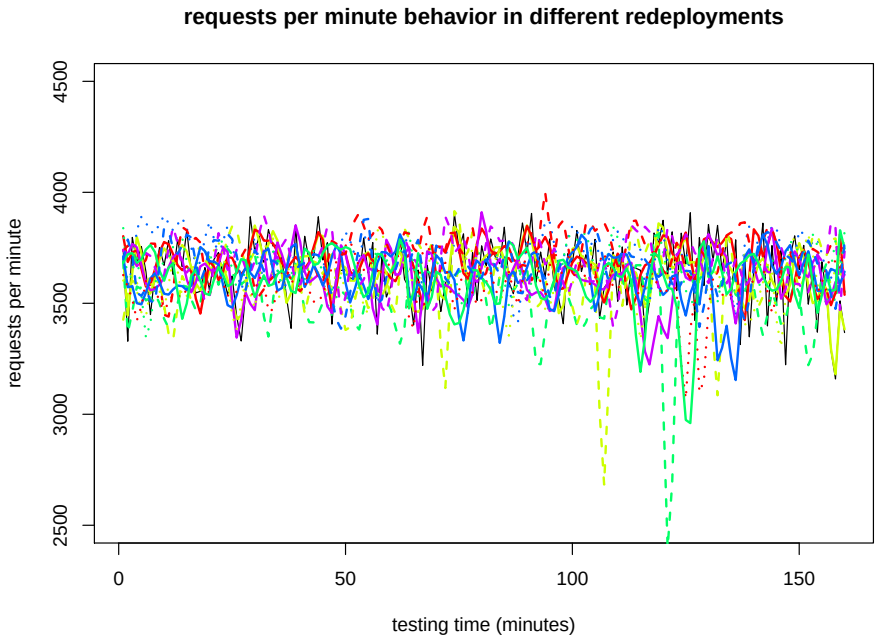

Figure A.7.: Requests per minute of load driver development in different deployments

Table A.1.: Median of cpu/usage_rate per test run (original unit: millicores)

Service \ Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	median value
carts	133,5	110	115	107	114	108	132	121	134	123	132	120	130	112	114	107	111	106	110	114
carts-db	44	37	38	36,5	38	37	44	41	45	40,5	44	40	43	37,5	38	36	38	36	38	38
catalogue	34	33	34	34	35	34	35	34	34	33	34	33	34	33	34	34	34	33	35	34
catalogue-db	26	25	26	26	27	26	26	26	26	26	26	25	26	25	26	26	26	25	27	26
front-end	300	295	298,5	301	309	300	304	296	305	307	302	295	306	302	299,5	301	302	302	311,5	302
load-test	375	365	374	377	379	368	371	372	372	365	369	365	378	363	370	369	372	364	375	371
orders	2,5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
orders-db	3	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
payment	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
queue-master	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
rabbitmq	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
session-db	3	5	5	5	5	5	6	5	6	5	6	5	6	5	5	5	5	5	5	5
shipping	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
user	4	4	5	4	4	4	5	5	5	5	5	5	5	4	5	4	5	4	4	5
user-db	3	3	3	3	3	3	4	3	4	3	4	3	4	3	3	3	3	3	3	3
zipkin	10	7	10	7,5	9	7	8	7	8	8	8	8	8	8	7	8	9	8	9	8
zipkin-cron	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
zipkin-mysql	55	35	52	35,5	52	34	57	34	59	34	55	34	57	34	53	35	51	35	52	51

Table A.2.: Variance of cpu/usage_rate per test run (original unit: millicores)

Service \ Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	median variance
carts	37	50	37	22	25	48	48	36	47	49	41	41	30	50	29	31	26	65	31	37
carts-db	7	5	4	4	5	7	8	6	7	8	7	7	5	7	5	5	5	10	5	6
catalogue	2	1	2	2	2	3	3	3	2	2	2	3	2	3	2	3	2	7	4	2
catalogue-db	2	1	1	2	1	2	2	2	2	2	2	2	1	2	2	2	1	4	3	2
front-end	110	96	108	140	90	152	132	136	115	125	109	216	91	151	137	183	115	470	176	132
load-test	124	138	188	182	149	225	193	213	175	178	187	367	129	225	205	276	214	800	215	193
orders	22	22	11	11	11	16	13	13	11	11	11	20	12	13	8	11	10	12	10	11
orders-db	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
payment	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
queue-master	25	6	5	5	5	4	3	5	4	5	5	4	5	6	3	5	6	7	5	5
rabbitmq	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
session-db	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
shipping	4	9	13	10	18	12	20	8	15	18	12	8	9	7	11	16	10	11	16	11
user	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
user-db	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
zipkin	299	433	718	339	834	429	476	323	344	320	413	416	476	466	566	467	278	694	600	433
zipkin-cron	2	1557	1533	1206	1343	1971	2232	2100	2335	2892	2170	1953	1412	1322	1203	2189	1845	1436	2295	1845
zipkin-mysql	27	362	899	438	632	501	1063	379	1344	198	1219	489	1034	418	1015	298	1725	372	1943	501

Table A.3.: Median of memory/usage per test run (unit: mebibytes)

Service \ Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	median value
carts	766,1	687,4	667,0	673,8	668,9	763,5	715,1	676,9	657,8	758,7	673,2	722,5	690,0	774,2	668,1	646,6	625,5	732,9	784,3	687,4
carts-db	162,0	156,4	156,3	162,1	162,9	160,9	159,2	159,4	161,7	160,5	161,1	160,2	164,3	159,2	162,7	159,2	161,7	157,7	160,4	160,5
catalogue	14,5	14,0	18,9	16,5	16,3	16,7	16,7	16,8	17,3	15,9	16,6	16,9	16,2	15,6	15,8	16,9	16,0	18,7	17,6	16,6
catalogue-db	203,5	203,1	239,1	440,9	333,2	289,1	258,5	355,3	286,5	265,9	251,0	379,1	279,9	249,7	257,1	289,0	291,9	265,6	261,9	265,9
front-end	111,5	112,0	129,2	129,2	128,2	126,7	123,9	128,7	127,0	125,1	119,0	128,2	127,5	121,0	124,0	126,3	127,3	127,1	125,0	126,7
load-test	39,1	38,9	42,3	60,6	53,2	51,3	49,7	56,8	51,8	49,0	45,9	56,9	52,7	48,7	50,9	51,3	52,1	53,0	51,2	51,3
orders	317,7	300,9	340,2	340,7	333,1	326,2	313,4	330,6	324,5	321,0	315,9	340,6	323,7	313,7	321,2	321,5	318,1	320,9	321,9	321,5
orders-db	37,2	48,7	42,1	69,3	63,8	62,9	59,2	64,5	63,3	63,2	56,8	65,0	63,9	59,2	62,6	63,0	63,6	63,5	63,3	63,2
payment	15,2	14,3	16,0	14,4	16,8	13,7	16,8	13,0	16,9	14,2	16,9	13,4	15,9	12,6	16,5	12,1	15,5	15,4	16,8	15,4
queue-master	301,9	257,3	295,0	225,8	291,3	256,1	267,6	259,9	298,9	257,0	288,1	256,6	270,4	246,5	290,3	245,5	283,4	245,7	294,1	267,6
rabbitmq	101,0	100,4	103,4	100,4	95,1	100,0	97,4	100,8	98,5	97,5	98,1	101,1	99,5	99,9	98,8	97,7	100,3	100,6	100,4	100,0
session-db	4,6	4,7	4,6	4,6	4,6	4,7	4,6	4,7	4,6	4,7	4,6	4,6	4,7	4,6	4,7	4,6	4,7	4,6	4,7	4,6
shipping	326,7	284,3	326,9	331,5	317,9	316,7	302,1	328,7	315,2	309,1	297,4	325,5	309,4	303,6	308,3	310,7	316,0	316,3	308,3	315,2
user	14,8	16,4	17,9	18,1	18,9	17,8	18,8	16,5	16,3	18,1	17,7	18,1	16,4	17,8	18,3	18,7	19,0	17,7	17,8	17,8
user-db	43,6	60,2	61,6	61,4	59,8	61,4	59,9	60,0	59,8	60,2	60,1	60,5	61,1	59,9	59,0	59,8	60,0	59,5	59,7	60,0
zipkin	1686,6	1494,4	1168,7	1540,8	1528,8	1502,3	1471,1	1504,3	1470,5	1502,1	1449,8	1480,2	1499,5	1466,7	1611,8	1421,1	1465,6	1393,5	1460,4	1480,2
zipkin-cron	40,9	293,4	211,2	208,7	181,7	217,1	213,5	283,3	233,8	322,5	203,7	280,1	233,4	277,2	200,3	258,9	151,0	195,6	200,1	213,5
zipkin-mysql	711,7	546,0	1009,3	477,4	1018,8	552,4	930,5	538,2	932,2	445,1	857,6	465,1	885,4	460,9	1083,6	466,2	1003,9	462,7	961,7	711,7

Table A.4.: Variance of memory/usage per test run (original unit: bytes)

Service \ Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	median variance
carts	7,E+13	9,E+14	9,E+14	5,E+14	2,E+14	2,E+14	1,E+14	3,E+14	1,E+15	2,E+14	6,E+14	8,E+13	1,E+14	1,E+14	3,E+14	4,E+14	8,E+14	1,E+14	5,E+14	2,62E+14
carts-db	1,E+15	1,E+15	1,E+15	2,E+15	1,E+15	2,E+15	1,E+15	2,E+15	1,E+15	2,E+15	1,E+15	1,E+15	1,E+15	1,E+15	1,E+15	1,E+15	2,E+15	1,E+15	1,E+15	1,45E+15
catalogue	9,E+10	9,E+11	4,E+10	1,E+10	1,E+10	1,E+10	1,E+10	3,E+10	1,E+10	3,E+10	3,E+11	2,E+10	1,E+11	5,E+10	8,E+09	4,E+10	6,E+09	1,E+12	2,E+10	2,72E+10
catalogue-db	9,E+10	2,E+12	2,E+10	2,E+14	1,E+15	9,E+14	3,E+14	2,E+15	8,E+14	5,E+14	5,E+14	2,E+15	8,E+14	3,E+14	3,E+14	6,E+12	1,E+15	3,E+14	5,E+14	4,68E+14
front-end	2,E+12	1,E+13	3,E+12	2,E+12	2,E+12	6,E+12	8,E+12	2,E+12	7,E+12	8,E+12	2,E+13	2,E+12	2,E+12	8,E+12	4,E+12	3,E+12	4,E+12	5,E+12	5,E+12	4,43E+12
load-test	2,E+13	1,E+13	2,E+13	1,E+13	6,E+12	6,E+12	9,E+12	9,E+12	6,E+12	1,E+13	6,E+12	6,E+12	8,E+12	7,E+12	1,E+13	2,E+13	6,E+12	2,E+13	1,E+13	8,68E+12
orders	1,E+14	5,E+11	2,E+12	4,E+12	2,E+13	3,E+13	2,E+13	1,E+13	4,E+13	1,E+13	4,E+13	1,E+13	2,E+13	1,E+13	5,E+12	3,E+12	3,E+13	2,E+12	7,E+12	1,19E+13
orders-db	3,E+12	1,E+14	3,E+11	2,E+12	2,E+11	6,E+11	9,E+12	2,E+12	2,E+12	3,E+12	4,E+13	3,E+12	5,E+11	2,E+13	5,E+11	4,E+11	2,E+11	2,E+10	2,E+12	1,71E+12
payment	4,E+10	2,E+12	3,E+11	2,E+12	2,E+10	2,E+12	2,E+10	2,E+12	2,E+10	2,E+12	1,E+10	2,E+12	2,E+10	2,E+12	3,E+10	2,E+12	2,E+10	3,E+12	1,E+10	2,62E+11
queue-master	2,E+12	1,E+14	7,E+11	2,E+14	9,E+11	2,E+14	1,E+12	2,E+14	6,E+11	2,E+14	2,E+12	2,E+14	4,E+11	2,E+14	4,E+12	2,E+14	1,E+13	2,E+14	5,E+12	1,36E+13
rabbitmq	3,E+09	3,E+09	2,E+11	2,E+10	7,E+12	2,E+10	8,E+12	3,E+10	3,E+12	3,E+10	5,E+12	2,E+10	6,E+12	3,E+09	2,E+12	2,E+09	7,E+11	2,E+09	1,E+12	2,73E+10
session-db	5,E+08	2,E+09	1,E+08	1,E+08	1,E+09	1,E+07	2,E+09	5,E+07	6,E+08	3,E+08	1,E+09	2,E+08	9,E+08	5,E+08	6,E+08	2,E+08	5,E+08	5,E+08	4,E+08	4,72E+08
shipping	6,E+11	2,E+11	1,E+12	6,E+11	2,E+13	3,E+13	2,E+13	1,E+13	4,E+13	2,E+13	4,E+13	1,E+13	2,E+13	2,E+13	8,E+12	5,E+11	3,E+13	1,E+12	1,E+13	1,23E+13
user	1,E+12	4,E+10	3,E+10	5,E+10	2,E+10	5,E+10	2,E+10	4,E+10	3,E+10	4,E+10	1,E+11	2,E+11	2,E+10	4,E+10	3,E+10	5,E+10	2,E+10	3,E+10	2,E+10	4,02E+10
user-db	2,E+12	4,E+11	4,E+11	3,E+11	2,E+12	3,E+11	3,E+12	9,E+11	1,E+12	1,E+12	2,E+12	5,E+11	1,E+12	2,E+11	6,E+11	6,E+11	4,E+11	6,E+11	3,E+11	5,83E+11
zipkin	1,E+17	6,E+16	7,E+16	5,E+16	5,E+16	4,E+16	6,E+16	6,E+16	7,E+16	4,E+16	4,E+16	4,E+16	4,E+16	4,E+16	8,E+16	3,E+16	9,E+16	4,E+16	4,E+16	4,83E+16
zipkin-cron	7,E+13	2,E+15	2,E+13	1,E+16	8,E+15	7,E+15	7,E+15	8,E+15	5,E+15	1,E+16	2,E+15	2,E+15	5,E+15	3,E+15	9,E+15	1,E+16	5,E+15	7,E+15	4,E+15	5,21E+15
zipkin-mysql	1,E+16	1,E+16	2,E+17	2,E+15	1,E+17	5,E+15	1,E+17	4,E+15	1,E+17	2,E+15	1,E+17	1,E+15	1,E+17	2,E+15	1,E+17	1,E+15	1,E+17	1,E+15	2,E+17	1,41E+16

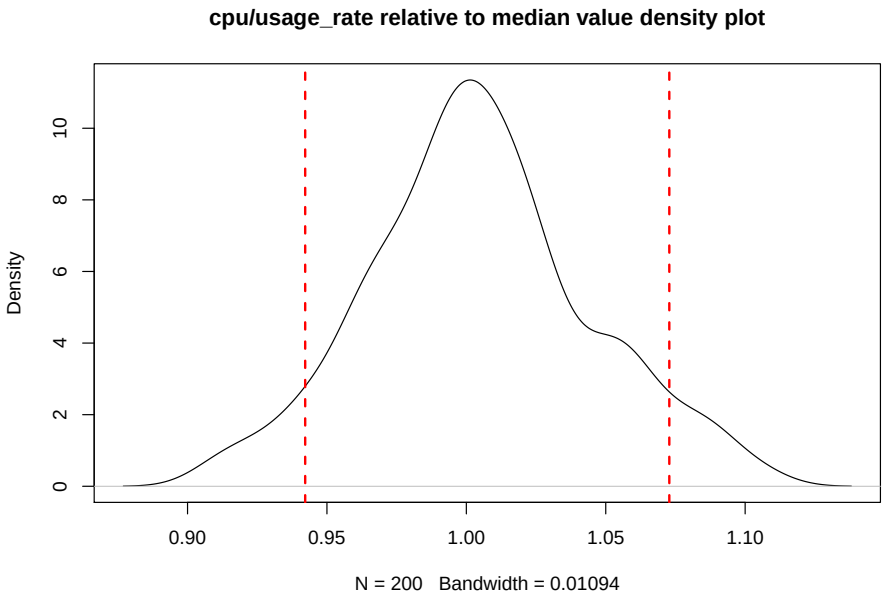

Figure A.8.: Cpu/usage_rate distribution relative to median value

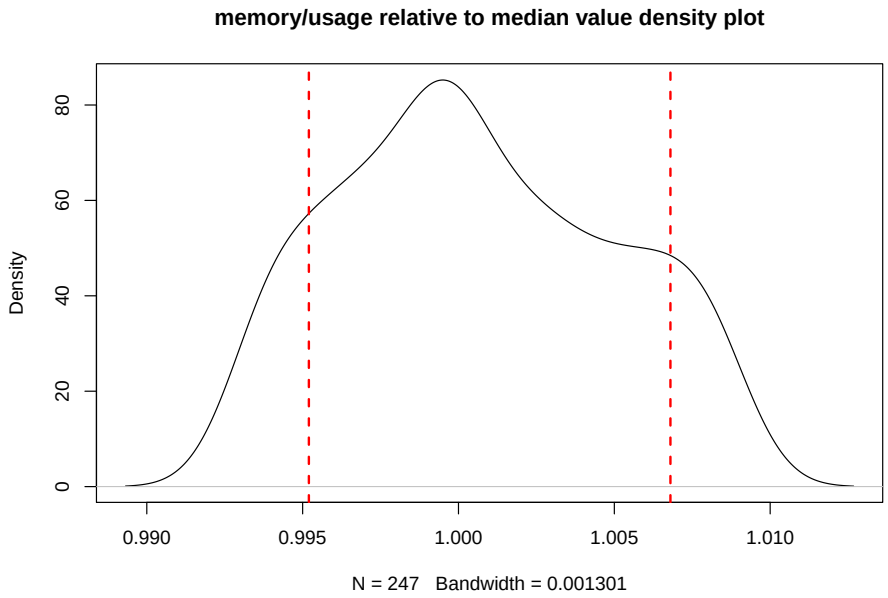

Figure A.9.: Memory/usage distribution relative to median value

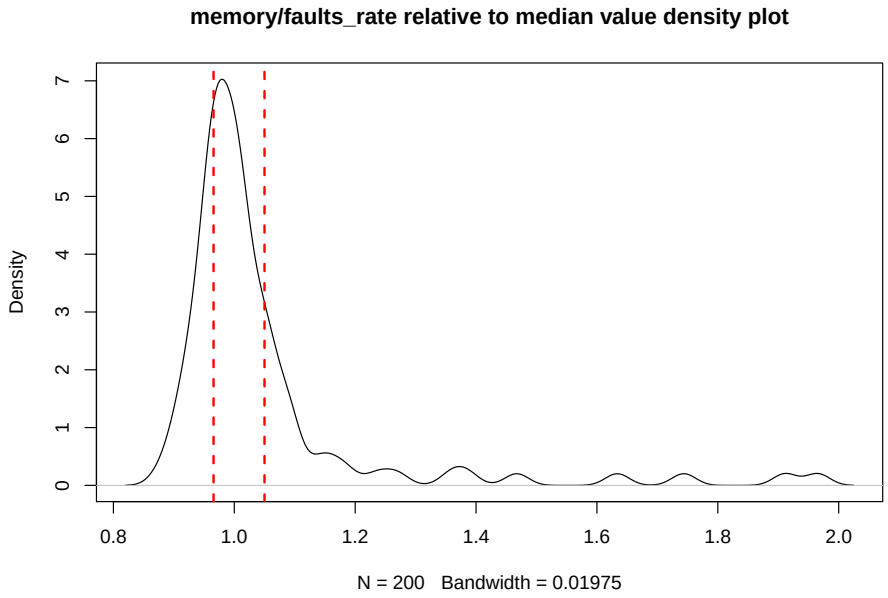

Figure A.10.: Memory/page_faults_rate distribution relative to median value

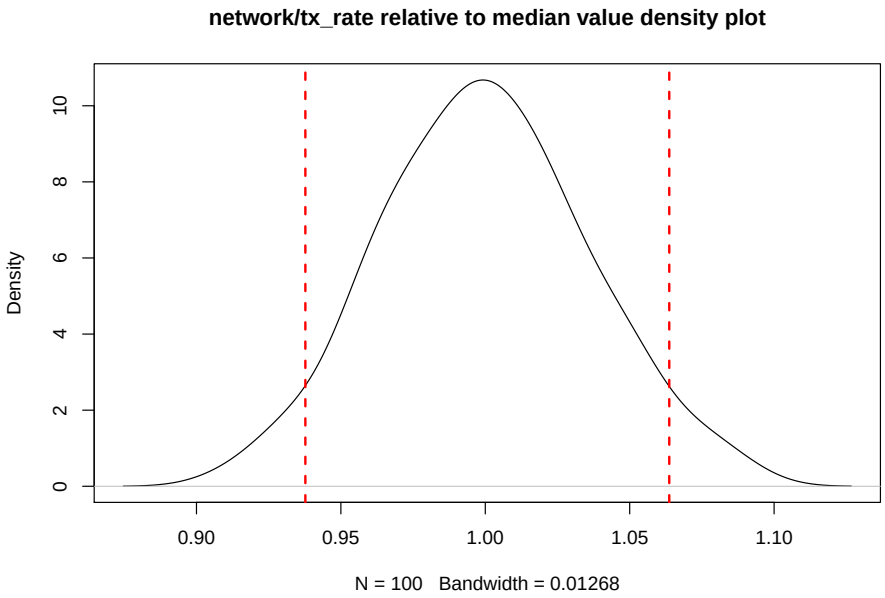

Figure A.11.: Network/tx_rate distribution relative to median value

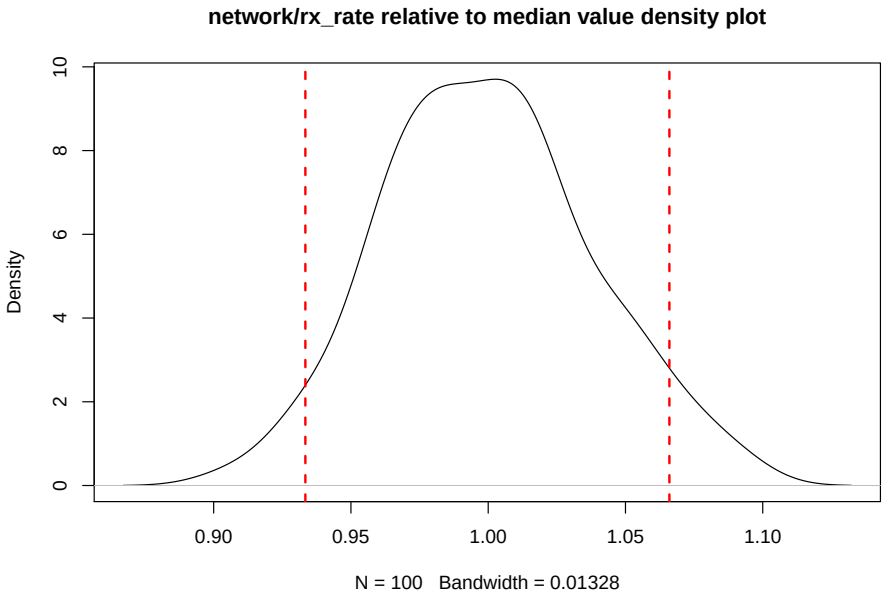

Figure A.12.: Network/rx_rate distribution relative to median value

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