Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 12345

# Automated Root Cause Isolation in Performance Regression Testing

Sebastian Vogel

| | |
|---|---|
| **Course of Study:** | Softwaretechnik |
| **Examiner:** | Dr.-Ing. André van Hoorn (Prof.-Vertr.) |
| **Supervisor:** | Dr. Dušan Okanović |
| **Commenced:** | December 16, 2016 |
| **Completed:** | June 16, 2017 |
| **CR-Classification:** | I.7.2 |

# Abstract

Testing of software is an important aspect of software development. There exist multiple kinds of tests, like unit tests and integration tests. The tests this thesis will focus on will be load tests, which are used to observe a system's behavior under load. The presented approach will use these load tests in order to observe and analyze the performance of a system, like e.g. the response times of methods. Next these observations are compared with those made on other versions of the system, in order to detect performance regressions, deteriorations in performance, between versions. Another goal of the approach will be to identify the root cause of the regressions, which is the source code change responsible for introducing them. By doing this, the task of fixing this problem will be made easier for the software engineer, since he has an entry point for the problem.

# Kurzfassung

Das Testen von Software ist ein wichtiger Bestandteil der Software-Entwicklung. Es existieren viele Arten von Tests, wie Unit-Tests und Integrationstests. Die Tests, auf welche sich diese Thesis fokussiert, sind Lasttests. Diese werden genutzt um zu beobachten, wie ein System sich unter Belastung verhält. Der vorgestellte Ansatz wird diese Lasttests nutzen, um das Betriebsverhalten eines Systems zu erfassen und analysieren, wie z.B. das Antwortzeitverhalten von einzelnen Methoden. Als Nächstes werden diese Beobachtungen mit denen verglichen, die auf anderen Versionen des Systems gemacht wurden, um Regressionen im Betriebsverhalten, wie Verschlechterungen des Antwortzeitverhaltens, zwischen den Versionen zu finden. Ein weiteres Ziel des Ansatzes wird es sein, die Hauptursache einer Regression zu identifizieren, welches die Quellcodeänderung ist, die für die Einführung der Regression verantwortlich ist. Dies wird es dem Software-Entwickler, der beauftragt wurde die Regression zu verbessern, einfacher machen dies zu tun, da er bereits einen festen Ansatzpunkt geliefert bekommen hat.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

**DAG** directed acyclic graph

**IDE** Integrated development environment

**SLAs** service level agreements

# List of Listings

# List of Algorithms

Chapter 1

# Introduction

Performance is an important aspect in software development. Amazon stated, that every 100 milliseconds of latency cost them 1% in sales [Nat08]. Similarly, when Google [Gre06] made a test and tripled the number of search results displayed according to user feedback, the searches within the test group dropped by 20%. The reason for this was the page generation taking 0.5 seconds longer. Google [VGG+13] also found out that, if an delay of 400 ms is added to every search, users will do 0.74% less searches. Since these studies show the importance of performance for the user, keeping the performance of a system in a good spot is an important aspect of software development.
The common way to evaluate a system, for either functionality or performance purposes, is to test the system. Next these evaluations can be used to make changes to the program, for example get a certain functionality to work the intended way or to increase the performance. A system can be tested in multiple ways, depending on what the objectives of the tests are. Figure 1.1 provides a scheme of an usual software development process. The first stage is to gather the requirements for the software, on which a design will be based. Next the designed software is implemented and tested. Here two kinds of testing are commonly done.
The first is the normal testing, which includes unit tests and integration tests. In unit tests only small parts of functionality are tested as a small unit. The usual goal for these tests is to check whether a single functionality works. This makes it a very short type of test. In Heger et al.'s [HHF13] case, they are also used to measure the performance of small functional blocks. One of the problems they encountered was, that a lot of the tests were too small and had more overhead than actual runtime. Integration tests on the other hand test, if functionalities or modules properly work together. These tests take more time, because they observe interactions, which is more complex, rather than to watch one functionality irrespective of the other parts of the system.
The second type of tests are staged tests, where the whole program is run in a staged testing environment. Tests done in this environment are usually long-running. Here usually load tests, performance tests or stress tests are done. According to Jiang et al.

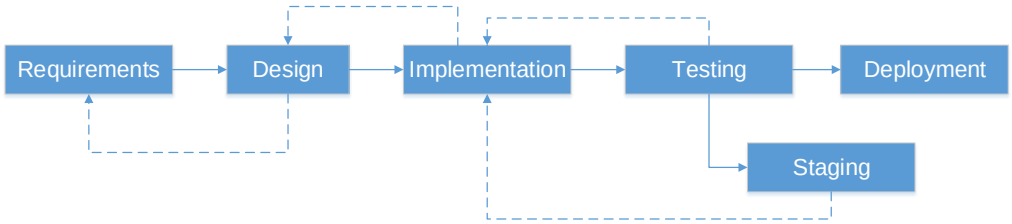


**Figure 1.1.:** Rough scheme of an usual software development process

[JH15] these are defined as follows: Load tests want to test the system under load of concurrent usage and observe their behavior. Performance tests are used to see, how the performance of the system as a whole behaves. This can be tested with various configurations for performance data on each configuration. Stress testing has the goal to test the system under extreme conditions, e.g. very high constant load or the missing of a stable connection to the Internet for smart phone apps.
The last step of the software development process is the deployment of the software into the production environment. During each step, if errors are found or complications occur, it is usually possible to take a step back to try another approach or fix the problem. Performance testing is especially important for the continuous integration environment. Here users have a direct comparison to the last version after a new version is deployed. Every new version is usually going through a series of automated tests in order to evaluate its functionality. One of the tests that can be automated are load tests, which put the system under artificial workload over a long period of time. They can be used to estimate, how the system will behave during production, when it is used and the load is for long times at the set level. This information can then be used to check if the application can behave correctly when under a load, that was set in service level agreements (SLAs) as the expected load, that the system has to handle.

Performance can also be measured on different versions of a system and compared, in order to see if the performance improved, remained at the same level or got worse. Unintended performance drops from one version to another are called performance regressions and can be very costly. Heger et al. [HHF13] stated as two characteristics for performance regressions: They are hard to remove and their removal is more costly the later they are detected. Therefore it is important to detect and fix these regressions as soon as possible. In order to fix these problems, a software developer has to detect the cause of the problem first. In a large program with a multitude of commits between software versions, this task gets harder. There exist some approaches, that focus on automatically finding these performance regressions and isolating the root cause, like Heger et al.'s, but there is no existing one, that concentrated on using load tests to do

this. The approach presented in this thesis takes Heger et al.'s approach and applies it to load testing in specific. The goal of the approach is to use data obtained from load tests, to evaluate the performance, find regressions and search for their root cause.
This Thesis adds to the automated tests of a continuous integration pipeline, with the goal of using already existing load tests in order to automatically detect regressions and their root cause. Regressions should be detected as early as possible. This is achieved by regular gathering of data, e.g. every time load tests are run on a new version. After detecting a regression the root cause of the regression has to be found. Giving this information to a developer makes it easier for him to remove the regression, since he already knows where to look for it. In order to reach this goal the approach uses two steps: A Bisection algorithm, in order to find the commit, which introduced the regression, and a call tree analysis, in order to identify, which method and class are most likely at fault.

## Goals

There are three major goals for this thesis. The first is to explain the foundations on which the thesis is based on, e.g. load testing and performance regressions, and how similar approaches, like the one used by Heger et al., work. The second one is to introduce the approach to automatically detect regressions and their root causes using load testing. Here details will be given on every step of the approach. The last major goal is to evaluate the approach by using a proof-of-concept implementation. The points, that should be evaluated are the reliability and scalability of the approach. Additionally possible improvements, that could be done for the approach are shown.

## Thesis Structure

The thesis is set up as follows:

**Chapter 2 – Foundations and related work:** This chapter serves as an entry point for the thesis and discusses the foundations it is based on. For this reason related work will be also presented here, like performance unit testing, regression benchmarking and performance evaluation methods.

**Chapter 3 – Approach for/to..:** This chapter describes the approach of the thesis to performance regression testing and root cause isolation. As a part of the approach used algorithms can be found here.

**Chapter 4 – Evaluation:** An explanation on evaluation method and results will be given in this chapter. Additionally the research questions will be stated and answered.

**Chapter 5 – Conclusion:** In this last chapter the results will be summarized and potential future work will be discussed.

Chapter 2

# Foundations and related work

This chapter will start by presenting the underlying technologies this thesis and the proof-of-concept implementation will be based on together with related work to the approach. First, testing will be described in general in section 2.1. The following section 2.2 will describe load tests, which will be used during this thesis. Afterwards a brief introduction to the related performance unit tests will be given in section 2.3. In the following sections 2.4 and 2.5 performance regressions and methods for performance evaluation will be shown. Section 2.6 will introduce regression benchmarking, another related topic. During section 2.7 root cause isolation will be presented. Next the software development style of continuous deployment will be discussed in section 2.8. At last in section 2.9 the tools used during the thesis will be briefly presented.

## 2.1. Testing

Testing is an important step during software development. It is usually the last step before deploying a developed program into production, as it can be seen in Figure 1.1. During this step the implementation of a software is tested for errors. This is done in order to make sure, that a system fulfills all SLAs and works properly. As stated by Myers et al. [MSB11] this step of software development takes up more than 50% of the budget of a software development project. Patton and Ron [Pat01] stated that the time effort, that goes into testing grew over the years.
Myers et al. divide testing into five types of tests. The first type they state are module tests, more commonly known as unit tests. These types of tests tend to be of a short nature and test if a small part of the code functions properly. Usually a set of input data is given to a method or function and it is tested if the returned results are equal to the expected results from the given input set. A special usage of these types of tests is shown in section 2.3. The second type Myers et al. state are integration tests. Here it is tested,

if the modules of a system work together the intended way. This type of testing is similar to unit/module testing and if the module testing is done incrementally it can be part of it. The third type of test are function tests. These are used to find discrepancies between what the customer expects the program to be able to do according to the agreements and what it is actually able to do. This step can be used to test, whether functions are missing. Another type of testing is acceptance testing. Here it is tested, whether the program is according to contract. This is usually not done by the developers, but the customer in order to check, that he got what he ordered.
The last type of testing is system testing. These tests have the widest variety with 15 different types of tests according to Myers et al.. This type of tests checks, if a system as a whole meets set requirements by the customer. This can for example include security tests, where the security of a system is tested, and performance tests, which tests how the system performs. Myers et al. state that this is only possible, when there are requisites to be met, as the functionality of the whole system is already tested during function tests. So according to them, whether these tests are done, depends on the contract.

Patton and Ron [Pat01] divided testing into similar types, but there are some differences. They define system testing more like function testing of Myers et al.. The types of tests, that were defined as system tests by Myers et al., were classified as specialized tests. This also includes performance tests, like described above, and stress or load tests, tests which test the system near or above the boundaries, that were set as required capabilities of the system. While stress and load tests are the same for Pattron and Ron, other literature differentiates. Gheorghiu [Ghe05] defines load testing as having concurrent high load on a system near the maximum of what it can take. The goal is not to break the system, but have it running for a long period of time. Stress tests on the other hand have the goal of breaking the system by applying more load than the system can handle. It is used for the developer to see how the system reacts to failures.

## 2.2. Load testing

Load tests [JD 07; Wey98] are a form of performance tests, that are used to measure the performance of an application under conditions, that should be similar to the target environment or higher. Doing this is one way to check, whether the performance of a program is in accordance to SLAs. Another benefit is, that the developers are able to predict the behavior of the system under the stress of production, in order to evaluate, whether the performance is good enough or has to be optimized. This originates from the fact, that data about performance, such as response times, throughput, resource utilization can be measured. Load tests are usually done with the help of tools, e.g.

JMeter [1], which send a defined number requests to an application in order to simulate multiple users using the program at the same time. Aside from performance it is also possible to observe the behavior under stress and stability, reliability and availability of the system.

Jiang et al. [JH15] described the process of load testing to consist of three steps. The first step is to design the load test. For this the objectives, that one wants to achieve with the test should be clear. This step includes the decision on what load is to be used. If the tests goal is to see, how the system handles a certain error, a fault inducing load can be chosen. If the fulfillment of SLAs is to be evaluated or how the system will behave under the expected usage, a realistic load similar to the stated requirements should be chosen. Other points, that should be decided on during this phase include length of the test run and whether to optimize the load depending on the technique used to generate it.
The second step is to execute the load test itself. This includes setting everything up, generating the load, execution of the tests and monitoring and collecting data from the tests. The test execution can be done according to three types of approaches. The first is to use live-users, who manually generate the load for the system. The second is to use a driver, that automatically generates the specified load. The last is to use special platforms, to deploy the tests into and execute them, that behave in an intended way.
The last step is to analyze the obtained system behavior data. This can be either to check against threshold variables, like the ones set by the SLAs. Another option is to see, whether a known problem still exists or if it was solved. Another option is to detect anomalous system behavior. This can be used to look for unknown problems.

Another important aspect is to automate tests. This is used to a large extent in continuous deployment environments, like described in section 2.8. Here tests are usually fully automated and run without manual inputs. Under these circumstances load tests also have to be automated. Bayan et al. [BC08] presented an approach to automate load testing. In their approach they automatically generated load for a system. They first identify what inputs influence the resource of a system, that is to be tested, like memory usage or response times. Next they automatically configured a controller, which is responsible for generating load in order to get the tested system to the desired load level.

One of the areas where this type is very common is web development [BD99] [Men02] [DGH+06]. Here HTTP requests are created and sent to the application in order to simulate users. One goal here is to simulate a 'virtual user' that behaves as close to how a real user would as possible. Examples for this would quitting, when the response time of the website is too high, and taking time to think between requests. During a load test

[1] http://jmeter.apache.org/

multiple 'virtual users' are to be simulated, since the test should achieve a load similar to what is expected later, when the web service is running.

## 2.3. Performance Unit Tests

Heger et al. [HHF13] propose using performance-aware unit tests. These tests had the goal to gather more valid data on performance, than normal unit tests. Usually, unit tests are rather short and test only a small part of functionality. This leads to low runtime of the tests. Heger et al. mentioned, that this low runtime of normal functional unit test, with often more overhead for preparing and launching the test than time needed for the test itself, would falsify the gathered performance data. For this reason they implemented performance-aware unit tests. These unit tests are designed with test length in mind, in order to be able to gather more valid performance data. The length of test can be extended by choosing the right coverage and granularity during test design.

Bulej et al. [BBH+17] propose to create a new category of unit tests, the 'performance unit tests'. Similar to how usual 'functional unit tests' test components for whether their functionality is working correctly, these performance unit tests would test, whether the performance of components is within acceptable parameters. In order to evaluate this correctly three challenges were stated by Bulej et al. that had to be considered, when creating performance unit tests. The first challenge is to define what *fast enough* means for a method. The easiest approach would be setting an absolute time, which defines for a specific test whether the test failed (the test took longer than the time set) or was successful (the test finished faster than the time set). But this approach has some flaws. The first one being defining the time for a single method on how long it should take. Another flaw comes with the platform the test is run on, since times can differ vastly there. Thus they propose the requirement to be not absolute.
The second challenge they state is the implementation of the unit tests. Here they propose to tackle the test implementation similar to how one would create a microbenchmark and be aware of common mistakes that can be made when creating these. The third and last challenge is the execution of the tests. When executing functional unit tests, compiling and running them in parallel will not matter. For performance unit tests this would be fatal, since it would influence the execution times of the tests. Other influences like garbage-collection can also influence this time. Their solution to this challenge is to execute these tests multiple times and evaluate the test conditions in a probabilistic manner.
Another work from Horký et al. [HLM+15] proposes using this new kind of unit tests in order to increase the awareness for performance of software developers. They propose

using performance unit tests in order to gather data on method execution times. In their approach they use a workload generator, which delivers dynamic input for the performance unit tests, which allows to scale the time the unit test takes. This is used in order to dynamically generate and acquire data when the developer wants to access it, similar to Javadoc. The performance data should then be displayed in a similar way to Javadoc as well. By doing so the approach tries to show information about method performance to the developer during development, raising the chance of the developer making decisions based on the software performance.

## 2.4. Performance Regressions

As a performance regression Heger et al. [HHF13] defined the "[...] significant increase of the response time of a tested method". In their context the increase of response time happened from one commit of the project to another, not during a single test run. The cause for the increase in response time is undefined in this definition and as such can be anything. This is opposed to another definition [ZAH12] found in literature for 'regression', which defines it as the reintroduction of an already solved problem (bug) into the code. An older definition [NTY07] defines it as a bug, that causes features to stop working after a certain event, like a patch. During this thesis the definition stated by Heger et al. will be used as definition for performance regression, as their work serves as a basis for this thesis.

## 2.5. Performance Evaluation Methods

Georges et al. [GBE07] presented different methods to evaluate the performance of a Java system. They analyzed the methods in regards to two different aspects: data analysis and experimental design. For data analysis only a small minority used confidence intervals, which was also the approach Georges et al. recommended. Most of the approaches used a single number to describe the performance. This number was either the computed average or median over multiple measurements or a specific extreme taken from the runs, like best or worst. An example for the median being used can be found in [AHR02]. Here the median of 10 runs was used to get a representative data set. In [BGK+02] the average over three runs of a benchmark was used. While most approaches, which use a single number for performance, use the best run, the SPECjvm98 [Cor08] approach included both the best and the worst run. Exceptions that can be found here include [BM03], where the second-best method run was used and [BOP03], where only one test run was done and the resulting time returned.

The second aspect, experimental design, generally defines how the data is acquired. This includes factors like using one or multiple VM invocations and implementation, the number of benchmark iteration and how the program was compiled. An example here would be [MH06], where multiple benchmark iterations were run on a single VM invocation on the same system with the same VM implementation. Additionally, they compiled all methods before the benchmark and fully cleared the heap using garbage collection, in order to have as few random factors as possible influence the measurements. Another consideration that has to be made is, when to take the measurements. Arnold et al. [AHR02] differenced between steady-state performance and start-up performance. The difference between these two types, is that start-up performance includes class loading, which may influence the performance of methods negatively. While for short-term programs this may be of importance, long running services will not be influenced for most of their runtime. This implies, that for short programs measuring the startup performance would return more valid results, while for application that have long runtime, excluding the startup-performance and concentrating on steady-state makes much more sense.

## 2.6. Performance Regression Benchmarking

This approach developed by Kalibera et al. [BKT04] uses benchmarks in order to find performance regression. It is a specialized application of already existing benchmarks in order to measure performance. The requirement they state as a base for the benchmark is, that it has to be fully automated. This automation includes execution, data gathering and data analysis. The execution is the easiest to automate. For data gathering, the warm up period of benchmarks has to be ignored and only steady state has to be captured. The last point has to detect regressions in a quick and reliable manner. Therefore, they propose using small enough benchmarks, that are able to be run daily. They divide benchmarks into *simple benchmarks*, that test a single isolated feature with artificial workload and *complex benchmarks*, that test a number of features with real-world workload. As the second group is not as easy to interpret, Kalibera et al. use only simple benchmarks. Additionally they try to minimize the interference of e.g. the system on the benchmark results and run the benchmark multiple times in order to gather enough data. Since the results from even one simple benchmark will vary when run multiple times, they propose using standard statistical tests for comparing samples from a normal population to compare the sets of different versions.
In later works Kalibera et al. also pointed out, that random initial conditions will influence each run of the benchmark. In [KBT05] they describe how to measure the influence of these random initial conditions using multiple test runs and how use this information to calculate the precision of a benchmark. This information can then be

used to evaluate, how reliable regressions are detected using these benchmarks. In [KT06] they describe how random effects of compilation, a subcategory of these random initial conditions, influence the performance. An example for this is how a compiler generates a random name for symbols in anonymous name spaces. These names are different every time and as a result the linker will place them at different places within the binary.

## 2.7. Root cause Isolation

Heger et al. [HHF13] introduced an approach, which used existing unit-tests to test performance of a project and use this data to check for performance regressions. The idea for this approach was, to compare the performance of the project frequently, e.g. weekly, with the last measured performance. Should a regression be found out, the next step would be to identify possible root causes for the regression. This was done with the goal in mind to reduce the time needed to fix the problem.

The regression detection method used by Heger et al., was based on the statistical approach to find performance regressions proposed by Goerges et al. [GBE07]. The idea for this approach is to use the mean of measured response times and calculate the confidence intervals over these values. These confidence intervals could then be compared in order to identify possible regressions. This method was proposed, since unpredictable outside influences can alter response times and by using statistics and a large pool of measurements these become less relevant. For more details on this approach check the chapter describing the approach 3.3. In Heger's case they used 50 executions of the unit tests as base to collect the data.

After detecting a regression Heger et al. continued by identifying the faulty code piece, that introduced the regression. This was done in two steps. First, they used the GIT bisect function [GIT17] in order to find the exact commit, which introduced the regression. The GIT-Bisect function uses a bisection algorithm similar to a binary search. By marking commits as 'good' or 'bad' the first commit, that introduced the regression is singled out. For more details on this algorithm see 3.4.
After the commit is known, the faulty part of the source code has to be found during the second step. For this they started by extracting the call graph using the Kieker monitoring tool and running the unit test with full instrumentation. After knowing the structure of the call tree, they proceeded by extracting performance data on each method by running the test multiple times with partial instrumentation. By doing this they created an annotated call tree, with methods as nodes, that have all data on their performance as annotation. Next they compared the call tree from a commit before the regression with the one from the commit that introduced the regression. If the call tree

of a method changed, they marked it as root cause. If the call tree was the same, they looked if a method, that was identified as having regressed, contained another method, that regressed. The called method was then identified as the cause for the regression in the other method and the process was repeated till a method was found, that didn't contain regressed methods. An additional criterion was, that only methods, that were updated in the identified faulty commit were able to be selected as a cause.

The problem with the approach were the existing unit-tests. These tests are usually made to only test very small parts of a system. This results in execution times, that consist more of preparation for the test, than the actual test took place. These times are hard to compare, since the overhead is too influencing. This results in the need for unit tests, which have the goal to also have a size to be significant enough to evaluate the performance.

Approaches similar to Heger's often have the goal to heighten performance awareness of the developer during development. One of these approaches is the one by Horký et al. [HLM+15] presented in section 2.3. They used performance unit to obtain performance data, which they planned to show the developer in an Integrated development environment (IDE) similar to Javadoc. Another approach, that is similar, is the approach of Danciu et al. [DCBK15], which creates an performance model for Java EE components in order to predict their performance. This data is then provided for the developer, which enables him to see how changes will likely influence the performance. A third approach by Kalbarczyk et al. [KIZ+14] also tries to heighten performance awareness. Their approach was mainly tested for C, but some other languages are also supported. They search for performance differences between version and identify changed functions using a tool. This data is then given to the developer, to give him the option to fix the problem. They also integrated their tool into GIT, which makes it possible to run it on every commit.

An example for root cause isolation in other software areas, is the approach of Bellur et al.'s [BA07]. They try to isolate the root cause of failure in J2EE environments. For this they first monitor the system for failure symptoms. When they find some they generate call traces and a topology graph for the system. These are used to generate a probability for the occurrence of certain exceptions in a component. These exceptions are then inserted into a Bayesian Belief Networks model together with the information on how they interact. The output of the model then provides information about the root cause of a certain exception, which is used to help a system in self recovery.

## 2.8. Continuous deployment

Continuous deployment is a software development method, which, according to Humble et al. [HF10], uses a 'deployment pipeline' in order to create more stable versions of an application. For this every change to the code has to be built first, before being tested by multiple automatic tests. If this is successful for a change, it is committed into the version control system, like e.g. GIT. By doing so, since every change is tested, less errors are introduced. This also leads to multiple runnable commits on the version control tool, which makes choosing revisions for releases easier. Humble et al. proposed in an earlier paper [HRN06] four principles, that can be used as a basis when trying to use a deployment pipeline. The first one is, that every stage of the build should provide a working program. There shouldn't be stages, where the built software doesn't work. The second is to always deploy the same artifacts into every environment. Runtime configurations should be managed seperatly. The third principle states, that there should be several testing stages, that are fully automated together with an automated deployment. The last principle is to evolve the production line. If the assembled application changes, the production also should be changed accordingly to fit the needs of the application better.

Rahman et al. [RHWP15] defined continuous deployment as "[...] a software engineering process where incremental software changes are automatically tested, and frequently deployed to production environments", during their survey on continuous deployment practices of different companies. They also name benefits of this software development approach, such as lowering the cost of the development and improving both, the satisfaction of customers and the quality of the software. These benefits are also partly mentioned by Olsson et al. [OAB12]. Since the goal of this approach is the ability to deliver functionality of the software to the customer in a faster manner, a result of this is to be able to obtain information about the functionality faster, as the customer will use the feature earlier. The data will also be more realistic, as it is not obtained during preset tests, but from actual usage at the customers. This enables the development team to notice and tackle challenges the software encounters during usage earlier.

## 2.9. Tooling

The following subsections will describe the tools, that were used in this thesis to test the approach.

### 2.9.1. Kieker

Kieker[2] [VWH12] is a framework, which is able to monitor and analyze a system's runtime behavior. In order to monitor said behavior, Kieker provides measurement probes, which allow performance monitoring and also control flow tracing, and utility to analyze the obtained data. The framework is divided into two extensible parts, Kieker Monitoring and Kieker Analysis.

Kieker Monitoring's purpose is to monitor a system and log its performance data, e.g. execution times of certain methods. It currently supports multiple programming languages, like Java and .Net. The obtained data can either be written into persistent logs or can be streamed directly to a destination, where it is processed, using message protocols. The module is also configurable, making monitoring the whole system, or only certain parts of it possible. This configuration can additionally be changed at runtime, allowing for dynamic monitoring. For Java-based systems code-injection can be used in order to monitor any program.

Kieker Analysis analyses the obtained data, which can be done either directly when obtaining the data, or using the logs of the performance data. The module's pipes-and-filter structure makes it highly flexible and allows for plugins to be exchanged easily, in order to customize the analysis for a given system. The module also supports custom plugins making it highly extensible. Using the Kieker WebGUI or the Kieker Trace Analysis Tool also allows using graphical user interfaces to present the analysis results.

### 2.9.2. GIT-Hub

Git-Hub[3] is an online-platform for sharing code and version control. It allows programmers, who work on a project together, to obtain the newest code ('pull') or submit their own current version of it ('commit') in order to work on it together in an efficient manner from anywhere in the world. Each project is therefore stored inside a 'repository'. Within a repository every commit is saved, making it possible to go back to an earlier (or later) version of the code.

It is also possible to create 'branches' which allows the programmers to work on different versions of the repository at the same time. This works by creating a copy or snapshot, usually from the 'master' branch. This branch can then be modified and the code is updated by committing it. The branches and their commits are also saved in the

[2]http://kieker-monitoring.net/
[3]https://github.com/

repository. After the work is done it is possible to merge the branch with other branches or the 'master' branch, in order to introduce the changes or new feature there.

### 2.9.3. Locust

Locust[4] is an open source tool, that is able to generate load for a specified system. It can be used to do load testing. In order to use it, python code has to be written, that defines how users would behave. After this is done and given to Locust, it will automatically use the behavior code to generate users. The number of users and how many users are to be created can be defined at the start of each test run. Another benefit of this tool is its scalability. Since it supports distribution over multiple servers, it can be used to simulate millions of users at the same time.

### 2.9.4. Docker

Docker[5] is a software container platform. It provides the Docker container engine, which can be used to run a container on an operating system like Linux and Windows, on cloud platforms and many server systems. Containers are a virtualization technique and work by packaging software with everything it needs to run into an isolated environment. This makes containers more efficient and lightweight than Virtual Machines, which contain a whole operating system. It also ensures, that the software execution stays the same, no matter what underlying operating system is used. That also means, that developers, who work on the same project, can achieve the same test results regardless of their machine.

### 2.9.5. Kubernetes

Kubernetes[6] is an open-source application, that can be used to manage containerized applications. It is able to deploy the applications and also scale them, e.g. higher number of containers. Examples for the management functionality are, that Kubernetes restarts failing containers, like the ones that stopped responding, and is also capable of giving IP-Addresses to containers and balance the load between the containers.

[4]http://locust.io/
[5]https://www.docker.com/
[6]https://kubernetes.io/

### 2.9.6. OpenStack

OpenStack[7] is a platform for orchestrating clouds and is capable of controlling large scaled pools of compute, storage, and networking resources. It can be controlled by administrators using a dashboard, and users are able to get resources using a web interface.

[7] https://www.openstack.org/

Chapter 3

# Approach for/to...

This chapter will describe the approach. For this first an overview of the approach will be given together with a description on how it belongs into the software development circle during section 3.1. Next load testing, the method used for data acquisition, and how they compare to unit tests is discussed in section 3.2. Afterwards the Regression Detection Method is shown in section 3.3. The GIT-Bisect algorithm will be introduced in section 3.4 and the call-tree analysis is described in section 3.5. The last section, section 3.6 will give some information about the proof-of-concept implementation.

## 3.1. Overview

The approach was inspired by Heger et al.'s work in [HHF13]. The idea is to take the concept of testing for regressions and isolating their root cause and to use it for load tests. Figure 3.1 shows a rough sketch of the continuous integration life cycle. It is composes of five steps. The entry point of the cycle would usually be to first develop a system based on a contract. This would be equal to the upper steps of receiving feedback (or a contract in the first iteration) and developing based on it. After the development is done, the system or program can be built and compiled. The next step is to execute automated tests and inspect the system. This step is done in order to find and get rid of faults before deploying the software into production environment. After the system is deployed the customer provides feedback, which starts the cycle anew. The approach in this thesis would belong into the 'automated tests'-step of such a cycle. In order to fit into this stage, load tests, data gathering, data analysis and root cause isolation should be fully automated. In regards to a normal software development process as it was depicted in Figure 1.1 the approach would fit into the testing and staging steps. These steps should be fully automated in a continuous integration environment.

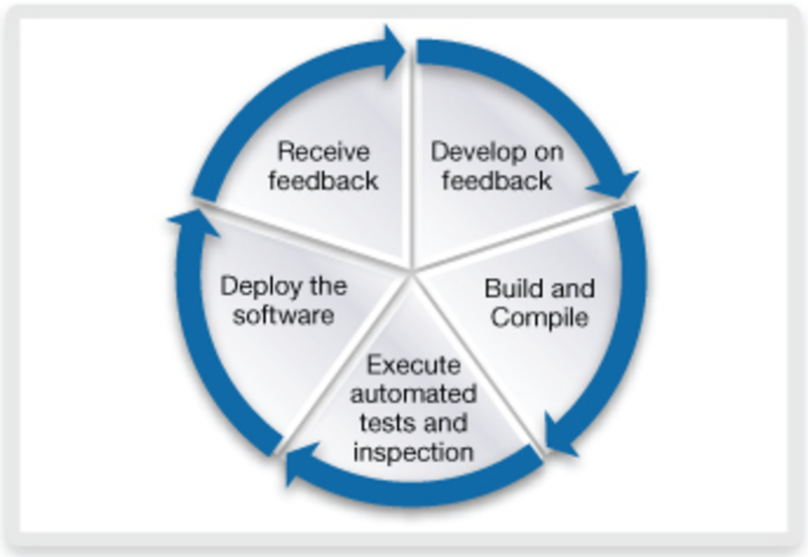


**Figure 3.1.:** Rough sketch of the continuous integration cycle. Taken from [Car17]

For the approach itself, first performance data is gathered while automated load tests are run (section 3.2). The data is then analyzed and compared with the data from previous versions in order to find possible regressions (section 3.3). After a regression is detected the root cause for it is searched. This includes a Bisection algorithm in order to identify the commit, that introduced the fault into the version control system (section 3.4) and a call tree analysis (section 3.5), which has the goal to find the method that is at fault for the loss of performance. After this analysis is done the information about the regression and the suspected root cause are given to the developer, who can then check the code and possibly fix the regression.
Figure 3.2 shows a overview of the approach. At first, in order to acquire data, load tests are run on different versions of an application, while a monitoring tool observes the application and gathers performance relevant data, like response times. After the load tests are done, possible performance regressions are detected, by computing confidence intervals over the measurements and comparing these intervals. If an regression was detected between to versions N and N+X, like depicted here, the root cause has to be identified. This is done first identifying the commit, that actually introduced the regression into the version control system. In order to do this, GIT-Bisect is used. After the commit is known, the call tree is analyzed, in order to additionally identify the methods responsible for the regression. By doing this, the developer gets more accurate feedback on where to start looking for the regression.

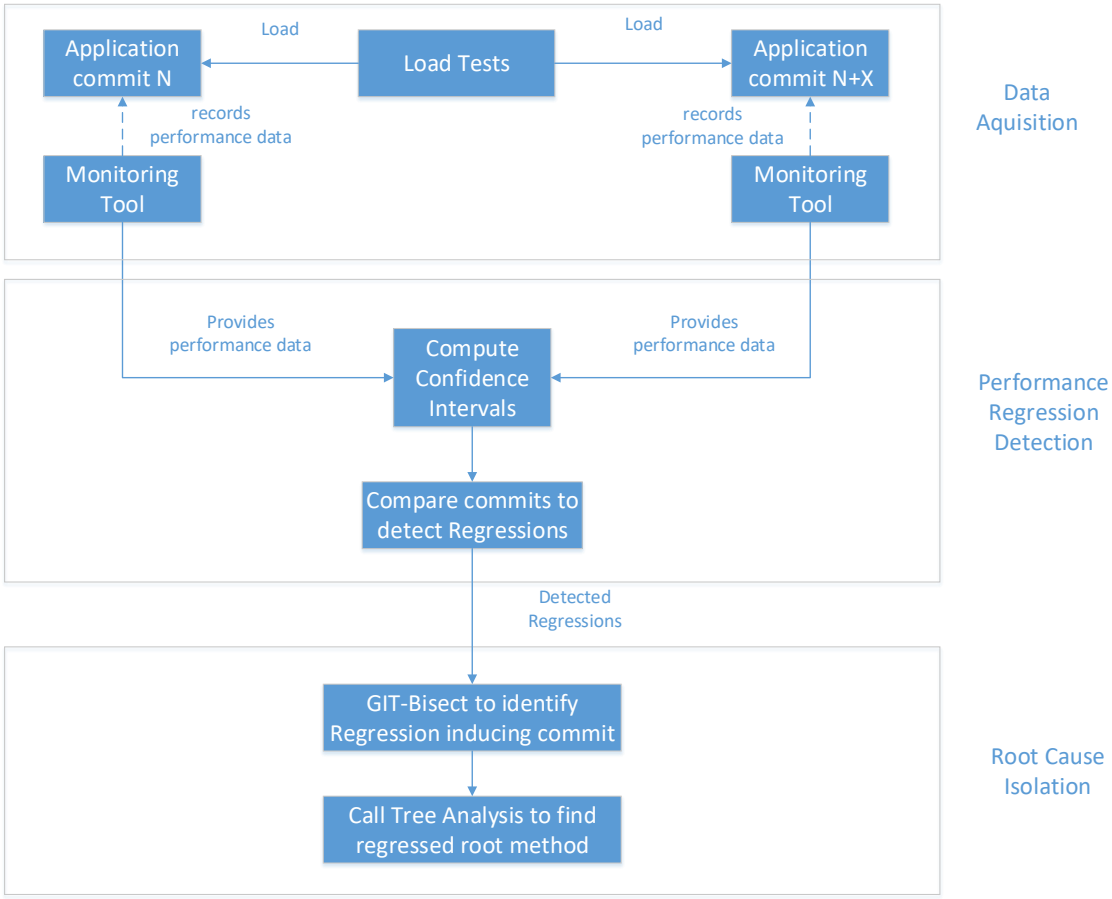


**Figure 3.2.:** Overview of the approach

## 3.2. Data Acquisition

In order to obtain valid and comparable data some key aspects are of importance. The data should be acquired under circumstances, that are as similar to each other as possible. For this, the same platform and tool setup should be used every time, when running the test. The same method for data collection should be used as well. Examples for this kind of setup can be found in the evaluation chapter 4.2.2 and in the appendix A. As the method for generating performance data, load testing was used in this thesis. Load testing are defined as putting a system under concurrent load over a long period of time according to Jiang et al. [JH15]. The goal of these tests is usually to observe the system's behavior under long lasting load. The load may vary for the tests depending on what exactly is to be observed. It can either be realistic and near the level of actual intended usage, or fault-inducing and far above the threshold given by requirements.

For further information about load testing see section 2.2, for information on how they fit into testing in general see section 2.1. As this is the key-difference to Heger et al.'s [HHF13] approach the differences between load and unit testing and what influence they have will be discussed in the following.

Load Tests vs Unit tests

Usually, load tests are used to investigate, how the system behaves under load over a longer period of time, while unit tests only test a small part of the application for a very brief period. From this two key differences can be derived. The first being which part of the system is tested. For load tests, this is a full component, if not the whole program, while unit tests usually test a single functionality. This means, that the interference from other parts of the same system may be higher in load tests, when compared to unit tests. The second, and more important difference, is the time period. It takes only a short amount of time for a unit test to complete. Load tests, on the other side, run for a long period of time, up to several hours. This second difference can influence the decisions made for implementing the other algorithms.
The approach by Heger et al. for call tree analysis included extracting more exact performance data on every method of a call tree by running the unit test multiple times, while monitoring only this specific method. For this approach, only one fully instrumented run of the whole load test was expected. Another difference is the number of measurements, that can be taken within one run. For unit tests, usually a method is called once to evaluate if the function was implemented correctly, meaning for multiple measurements the unit test has to be repeated a couple of times. In the case of Heger et al. they repeated each test 50 times. In load tests, methods are called more often over the course of time, which means in order to gather multiple measurements the test has to be run for a certain period of time as opposed to restarting. This is also important, since startup performance, like described by Georges et al. [GBE07], is not as interesting for programs that have long runtimes, like the ones load tests are usually done on. Instead steady state is of more interest. Another benefit of running the test longer, is that the number of measurements for steady state can vastly outnumber the ones for startup, which means that the startup values don't influence the computed confidence intervals as much.

## 3.3. Regression Detection

The regression detection algorithm used is based on the work by A. Georges et al. [GBE07], which was also used by C. Heger et al. [HHF13] The approach argues, that

sources of non-determinism are a given in a Java system. Examples for this could be differences in thread scheduling and how threads end up interacting because of it, when the Java Garbage Collector becomes active or even system interrupts. Because of this, Georges et al. suggest an approach that is based on statistic's confidence intervals over the mean of multiple measurements.

### Confidence Intervals

Confidence intervals describe an interval of values, that are defined as such, that the real value is with a certain probability within the interval. In the case of response times, this would mean, that the true, uninfluenced, response time of a method would, with a certain probability, lie within the confidence interval. There are different formulas for large and low numbers of measurements. In both cases first the mean $\bar{x}$ and standard deviation $s$ of the measurements has to be computed using the formulas

$$\bar{x} = \frac{(\sum_{i=1}^{n} x_i)}{n}$$

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

where $x_i$ is a single measurement and n is the number of all measurements. Next the confidence interval $c$ can be calculated depending on the number of measurements $n$. For large $n$ ($n >= 30$) the formula is

$$c = \bar{x} \pm z_{1-a/2}\frac{s}{\sqrt{n}}$$

With $\alpha$ being the significance level and $z$ the z-value. The significance level has to be chosen beforehand, and determines how much deviation from the mean is acceptable. The term $1-\alpha$ is called confidence level and is defined as the probability for the true mean $\mu$ being between the two confidence borders $c_1$ and $c_2$, which are computed using the above formula. The z-value is fixed for any given confidence interval and is usually taken from a table. If the number of measurements is small ($n < 30$) the formula

$$c = \bar{x} \pm t_{1-a/2;n-1}\frac{s}{\sqrt{n}}$$

is used for the confidence intervals. The value $t$ for this is from the Student's t-distribution and usually taken from a table.

Regression Detection (Comparison of alternatives)

Georges et al. [GBE07] introduce two methods for comparing two alternatives, or versions. The first is to compute confidence intervals for both alternatives and comparing these intervals. When these two intervals overlap, it is concluded, that the difference in the measurements may result of random factors, and that there is no difference in performance between the versions. However, if they do not overlap there may be an improvement in performance or a regression (depending on which one is the earlier version). This method is not completely accurate and has a small room for error. The second method, which is more accurate, is to compute the confidence interval for the difference of the measurements. For this, the mean and the standard deviation for the difference are computed using

$$\bar{x} = \bar{x_1} - \bar{x_2}$$

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}.$$

The confidence interval for large numbers of measurements is calculated using

$$c = \bar{x} \pm z_{1-a/2} s_x.$$

For a low number the z-value has to be substituted by the t-value, which can be approximated using the formula

$$n_{df} = \frac{(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2})^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}.$$

For this comparison, if the interval includes 0, it is concluded, that there is no significant statistical difference between the measurements. If 0 isn't included there is a regression/improvement detected.

When comparing more than two alternatives, Georges et al. [GBE07] proposed using the ANOVA method. This method wasn't used for the implementation, since two alternatives are compared in all cases.

## 3.4. GIT-Bisect

Overview

GIT-Bisect [Cou08; CS14; GIT17] is an implementation of a binary search provided by GIT. Its purpose is to find a commit, that introduced a fault of any kind into a GIT-

repository. GIT-Bisect can be run manually or automatically. After starting the algorithm, for both options, the user has to mark an initial commit as ‘good’ and another later commit, e.g. the current commit, as ‘bad’. For the algorithm, a commit marked as ‘good’ equals a commit where the regression hasn’t appeared yet, while the ‘bad’ commits contain the regression.

After appointing these two commits, GIT-Bisect will automatically select a commit according to its Bisection algorithm and pull it from the repository. If GIT-Bisect is used manually, the user now has to test the pulled version and tell the algorithm, whether the commit is ‘good’ or ‘bad’. In an automated setup a script or command can be given to GIT-Bisect, which will be run in order to evaluate the commit. Other options aside from ‘good’ and ‘bad’ are ‘stop’ in case you want to abort the search, or ‘skip’, which will result in skipping the current commit and GIT-Bisect selecting and pulling another one. This second one is especially useful if e.g. the selected commit contains a version of the source code, that can’t be built. The process of the algorithm pulling a commit and marking this commit, is repeated according to the Bisection algorithm, until the faulty commit is found.

### Bisection Algorithm

The Bisection algorithm, that is used in GIT-Bisect, in order to select the next commit, was first implemented by Linus Trovalds and later improved by Junio Hamano. The algorithm consists of three steps, that are used to prepare for the selection of the next commit to be tested, and the selection of the commit itself. At first it is important to note, that GIT-commits for a directed acyclic graph (DAG). This is true for any change history, since a later version can’t influence an earlier one, and a certain version derives from an earlier one. Figure 3.3 shows an example of a commit tree. Nodes are commits and are labeled with letters. The arrows represent which commits influenced which other commits.

The first step of the Bisection is to reduce the DAG, with the result that only relevant commits are contained. For this all commits, that are not ancestors of the (earliest) ‘bad’ commit or the ‘bad’ commit itself are removed. Then all commits, that are marked as ‘good’ and their ancestors, will also be removed. Figure 3.4 the previous example of the commit tree. Now the good commit A is marked as green and the bad ones, G and H, as red. The commits, that are ancestors of good commits, like U and V, or do not influence bad ones, like W, X and Y, are marked in a darker shade of blue. The next figure, Figure 3.5, shows the reduced Graph with only relevant commits. Here commits U to Y were removed due to not being ancestor of a bad commit, or being ancestor of a good commit. The commit labeled Z was also removed, as it isn’t the first bad commit.

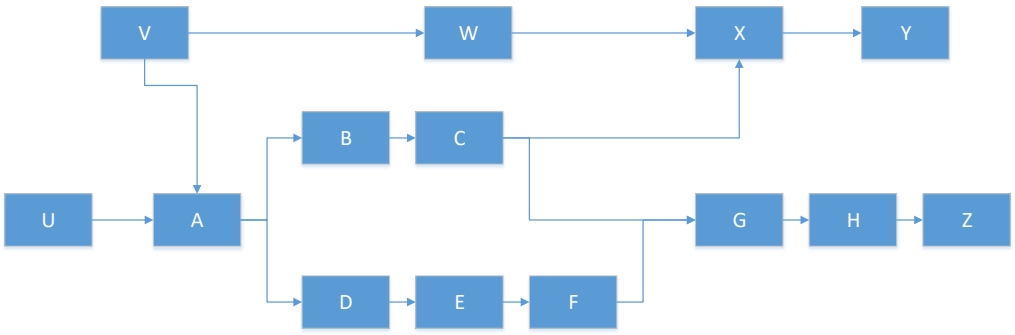


**Figure 3.3.:** An example for a commit tree represented as a DAG.

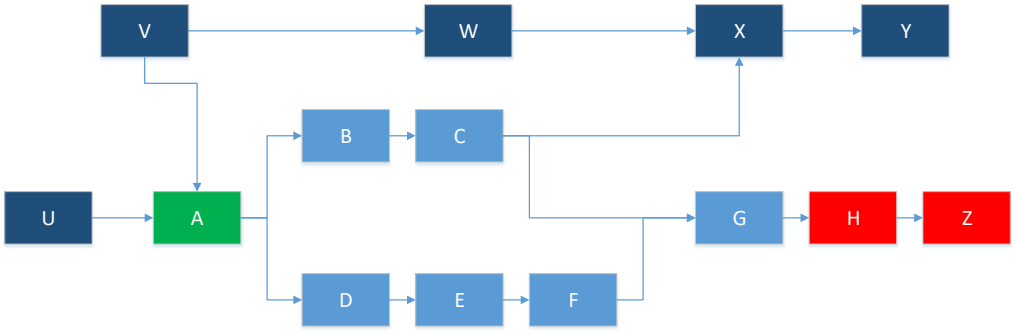


**Figure 3.4.:** The DAG of the commit tree with marked good and bad commits.

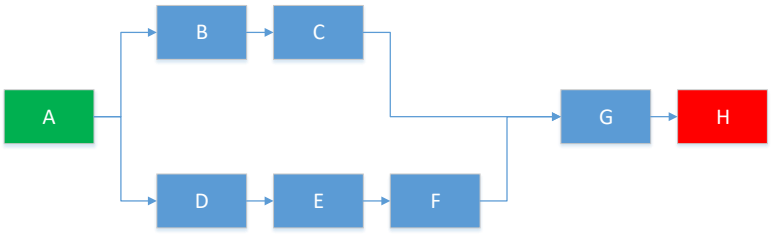


**Figure 3.5.:** The reduced DAG of the commit tree, which only contains relevant commits.

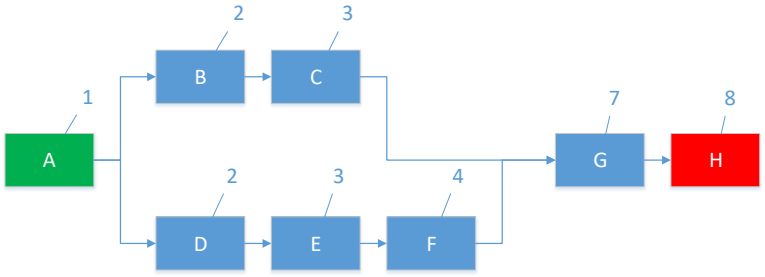


**Figure 3.6.:** The DAG, here with commits labeled with the number $A$

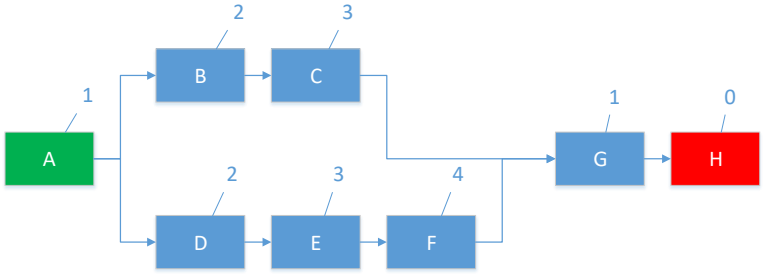


**Figure 3.7.:** The final DAG, this time with commits labeled according to $min(A, N - A)$.

The second step is to assign every commit a number $A$. This number is obtained by counting the ancestors of a commit in the DAG and adding one to it. An example for this is shown in Figure 3.6. Here every commit in the reduced DAG is labeled with the number of ancestors in the graph plus one. As example for A this would be no ancestors, so $A = 0 + 1 = 1$, while for H this would be seven ancestors, so $A = 7 + 1 = 8$.
This step is preparation for the third and last step, where each commit gets a label with its weight using the following formula: $min(A, N - A)$, where $N$ is the total number of Nodes in the DAG. This is shown in Figure 3.7. This DAG has the labels updated to the formula $min(A, N - A)$. As examples: A did not change since $1 < 8 - 1 = 7$, meanwhile later commits, which have many ancestors, like G, which has $A = 7$ and H with $A = 8$ were updated to $1$ and $0$ respectively. After the graph was prepared using the described three steps, the next commit which was labeled with the highest weight will be selected as the next commit to be tested. In the example from Figure 3.7, this would be commit F, which is labeled with a weight of $4$.

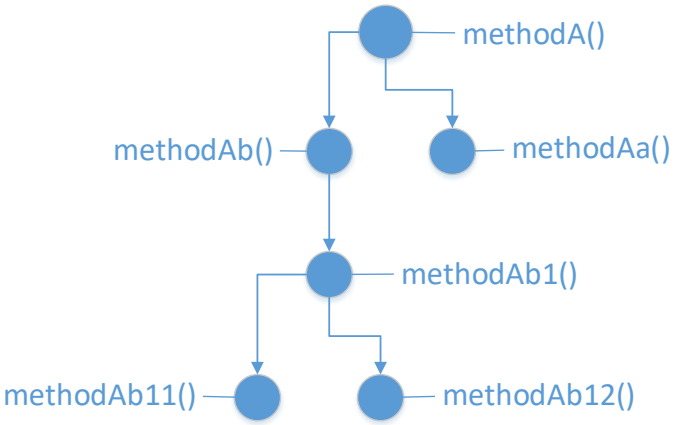


**Figure 3.8.:** An example for a call-tree.

## 3.5. Call-Tree Analysis

For the Call-Tree analysis, the call-tree has to be extracted first. For this purpose, OPEN.xtrace [OHH+16] was used. The OPEN.xtrace format is a model with the goal of allowing data interoperability and exchange for application monitoring tools. It is based on the fact, that most of these tools are extracting execution traces, which describe the control flow of method executions. This also equals the call-tree for a request. For this reason, the implementation of the OPEN.xtrace model is able to extract the call-tree from APM-tool data. One of the supported tools is Kieker, which was used in this setup to monitor the test-application. This means, by using OPEN.xtrace it is very simple to obtain the call-tree for a method from the Kieker monitoring data, which is why it was used. An example for a call tree is given in Figure 3.8. Here some methods are listed. Method *methodA()* calls two other methods, *methodAa()* and *methodAb()*. The first of the two again calls another method, *methodAa1*, which calls another two methods *methodsAa11()* and *methodsAa12()*.

After obtaining the call-tree, it has to be analyzed. For this purpose, the call trees of methods, that were found suspect of containing a regression are looked at. If a method directly contains another method, that was identified as possible regression, the cause of the regression is looked for in that method. The reason for this being, that the contained method, by having regressed, influenced the containing one, which resulted in slower Execution there too, since the response time of the contained method is part of the response time of the containing one. This only accounts for directly contained methods, since if a middle-tier method exists, whose response time was not influenced enough for

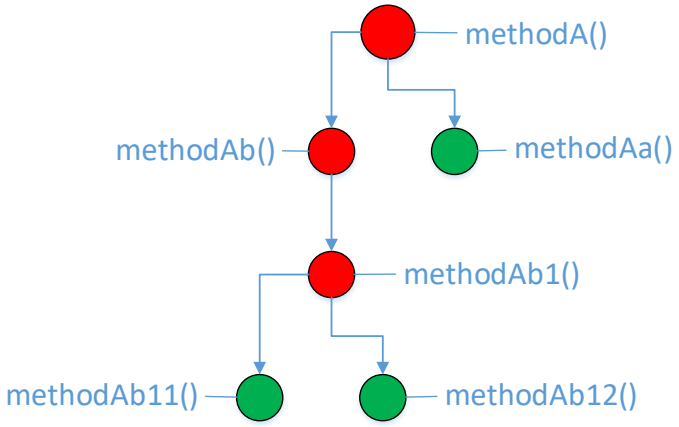


**Figure 3.9.:** The commit tree with regressed methods marked as red and nonregressed methods marked as green.

it to be detected as a regression, the regression in the lower method should not influence the upper method enough to have a high enough impact on performance. Another possibility is that the call tree of a method changed and that's how the regression was introduced. If the call tree of a method changed the method will be marked as a root cause of the regression. Figure 3.9 shows the previous call tree, this time with methods marked. Methods marked red have regressed, when compared to the previous version. Methods marked as green have not. In explicit, methods *methodA()*, *methodAa()* and *methodAa1()* have regressed. Since *methodA()* calls *methodAa()*, which again calls *methodAa1()*, the algorithm would conclude, that the most likely root cause of the regression would be *methodAa1()*, since this methods could have influenced *methodAa()* into being detected as regression, which in turn would have influenced *methodA()*. If *methodAa()* was not marked as regression, the algorithm would mark both *methodA()* and *methodAa1()* as likely root causes, since the regression in *methodAa1()* did not influence *methodAa()* enough to be detected as a regression. Because of that, it would also most likely not have influenced *methodA()*.

Another step is to look at the change-history of GIT if the commit, which introduced the regression, was found. Since GIT saves the information on which classes were edited together with the package name of the classes, in this step only methods that belong to classes, which are listed as touched, are looked at as possible root causes. This enables to exclude methods, that were detected to contain a regression, but were not touched in the commit, where the regression first appeared. Figure 3.10 shows the same graph as Figure 3.9, but with methods, which were edited in the detected faulty commit in form of a black dot. The methods influenced were *methodAa1()* and *methodAb()*. From

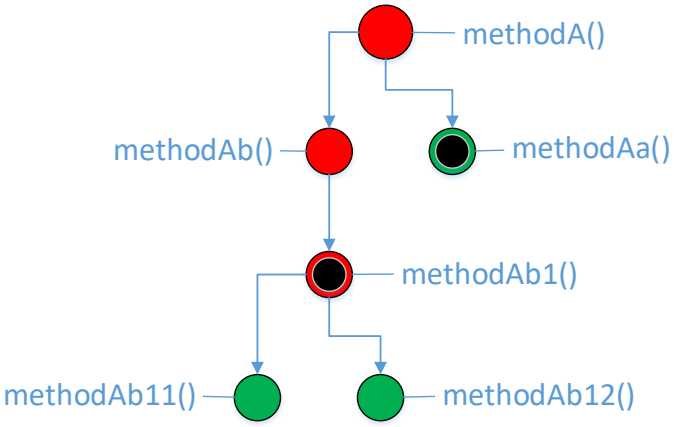


**Figure 3.10.:** The commit tree with regressed methods marked. Additionally methods influenced by the faulty commit are marked.

this the algorithm would select *methodAa1()* as the most likely root cause, since it was touched in the fault inducing commit and is the lowest method, that has regressed.

## 3.6. Implementation

The following will briefly describe some details on the implementation. This includes used technologies, data structures and some interesting code snippets.

### 3.6.1. Used Technologies

To start with, Java was used as programming language. The Java version used is Java 1.8. For call tree analysis Open.XTRACE [OHH+16] was used. Additionally Kieker[1] [VWH12] was used. For Git-Bisect the GIT implementation 2.13.0 for windows[2] was used.

[1]http://kieker-monitoring.net/
[2]https://git-scm.com/download/win

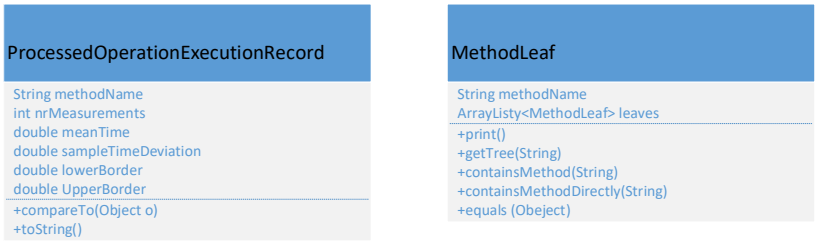


**Figure 3.11.:** Class diagrams of the model classes *ProcessedOperationExecutionRecord* and *MethodLeaf*.

### 3.6.2. Data Structures

There were two interesting data structures used during this thesis. Both are depicted in form of class diagrams in Figure 3.11. The first data structure is the class *ProcessedOperationExecutionRecord*, which contains the data for a method during a measurement. It holds the information about number of measurements, mean, standard deviation and lower/upper confidence borders. The class implements comparable in order to access methods in alphabetical order. The second class is *MethodLeaf*. This class is used to depict call trees in an efficient manner. It's *equals(Object o)* method was overridden in order to quickly being able to compare different call trees. The *getTree(String)* method allows to obtain the call tree of a called method and the methods *containsMethod(String)* and *containsMethodDirectly(String)* are used to check if the call tree, or only the root of the call tree, contain a certain method.

### 3.6.3. Interesting Code Parts

As mentioned earlier, Open.XTRACE was use to generate call trees, as it was easy to use. The following code piece shows how to easily obtain the Open.XTRACE data structures, that contain the call trees, from Kieker files:

```
LinkedBlockingQueue<Trace> traceListTraceOld = new LinkedBlockingQueue<Trace>();
String[] pTOld = { pathTraceOld };
TraceConversion.runAnalysis(traceListTraceOld, pTOld);
```

Afterwards the call tree can be extracted into the previously shown model class. Then the trees are compared with the suspicions, methods that were identified as regressions, in order to narrow the suspicions down to the root cause.

Chapter 4

# Evaluation

This chapter contains the evaluation of the approach. For this a proof-of-concept implementation was created. In the first section, section 4.1, research questions, that this thesis tries to answer will be introduced. Section 4.2 contains information about the evaluation methodology and how the experiment was setup. In section 4.3, the results of the the experiment will shown. At last, in section 4.4, the results will be discussed with reference to the research questions and threats to validity.

## 4.1. Research Questions

In the following the research questions of the thesis will be described. They serve as a basis for the evaluation. Since the evaluation is done on a proof-of-concept implementation, the questions will be answered using this implementation.

### 4.1.1. RQ1: How reliable is the approach?

This question is about the reliability of the approach. The reliability is here defined as, whether regressions are detected correctly and if, after a regression was detected, its root cause was isolated.

### 4.1.2. RQ2: How scalable is the created approach?

This is about evaluating the performance of the approach, by testing, how long it takes with different scales of projects. For this, different parts should be scaled, like size of the project, number of services, quantity of regressions and number of revisions.

### 4.1.3. RQ3: How can the approach be improved?

This question means researching how the approach could be improved in order to become better. This includes how the limits that were found in the previous questions could be altered for the better, along with how it generally could be improved.

### 4.1.4. RQ4: Is it worth to do a call tree analysis before GIT-Bisect?

This last research question overlaps with RQ3. The question here is whether doing a call tree analysis before doing GIT-Bisect could be an improvement. The goal of this question is to reduce the time needed for doing GIT-Bisect by straight up skipping commits, that are not of relevance to the method call-tree anyway.

## 4.2. Evaluation Methodology and Setup

The following subsections will describe which methodology was used for the evaluation and how it was set up.

### 4.2.1. Methodology

For the evaluation a proof-of-concept implementation for the approach was created, in order to be able to test it. In order to acquire data for the evaluation a test project was chosen and altered in order to create regressions. The created data was then given to the implementation in order to test its correctness and if the approach worked. The goal is to have the implementation find regressions and identify the correct root causes.

### 4.2.2. Setup

The methodology part roughly sketched out how the experiment was set up. As a test-project the Netflix RSS-Reader [net14] was used. Before running any tests, some Console outputs and fake-commits (e.g. add a whitespace and commit it) were introduced in order to create regressions within the program, that can be detected. It was set up as

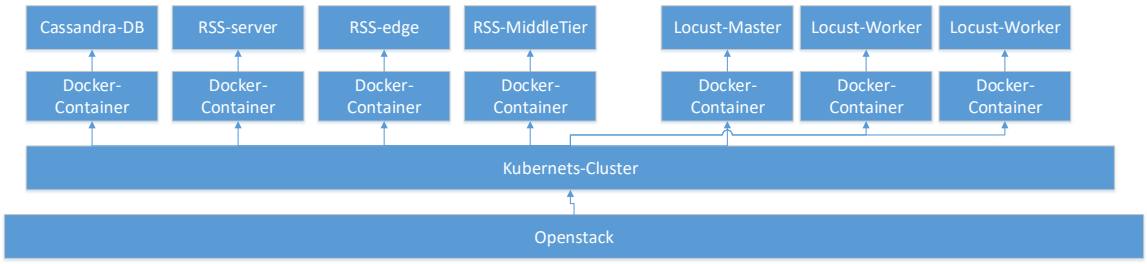


**Figure 4.1.:** Rough sketch of the test setup.

Docker[1] containers within a Kubernetes[2] Cluster, that is running on OpenStack[3]. A guide for this can be found in their wiki. For instrumentation of the program, Kieker [Kie17] was added into the Docker image. There is an existing load test for the program, which uses Locust[4] to send requests to it. For more information on these tools see section 2.9. Figure 4.1 shows a rough sketch of the setup. On the bottom is OpenStack with Kubernetes running on top. On top of the Kubernetes Cluster the four docker container run with the parts needed to run the RSS-Reader: Cassandra, the core of the server, edge and middletier. The middletier-part is instrumented with Kieker within the docker container. Additionally the Locust-Master container is running with two instances of Locust-Worker. This is generating the load. Aside from these depicted containers the kubernetes master node was also running for the cluster.

After the RSS-Reader was set up together with the Locust, Locust was started with 100 Users and 1 user per second as load test parameters. This test was then run for two hours. After two hours passed the test was stopped and the performance data gathered by Kieker was extracted. The data was then given to the proof-of-concept implementation in order to test it. The following listing shows which how regressions were inserted into the middletier-components by inserting code.

```
public class MiddleTierResource {
      [...]
       public void newMethod(){
             for(int i =0;i<100;i++)
                   System.out.println(i);
       }
      [...]
```

[1]https://www.docker.com/
[2]https://kubernetes.io/
[3]https://www.openstack.org/
[4]http://locust.io/

```
      public Response fetchSubscriptions (final @PathParam("user") String user) {
            newMethod();
            [...]
      }
      [...]
      public Response subscribe (final @QueryParam("url") String url,
      final @PathParam("user") String user) {

            for(int i =0;i<10000;i++)
                  System.out.println(i);
           [...]
     }
     [...]
}
```

As a remark, the proof-of-concept implementation only able to use parts of the data Kieker measures. Thus about 8%(up to 25000 overall measurements) of the data was taken and compared from the 2 hours of load testing (only a single .dat file of a Kieker folder at a time). This data still consisted of between 350 to 7000 measurements per method and was taken from around the same time into the test. For the most part used were: third set of measurements, about 20 minutes in and additionally compared second from last.

## 4.3. Observed Results

The results of the measurements varied vastly. Often methods, that were not touched came up detected as regressions. In some cases, unaltered methods were detected as regression, while other methods that were not touched were detected. An example for such a method was *com.netflix.recipes.rss.jersey.resources.MiddleTierResource.unsubscribe*. This method came up as regression often while not being altered.
The measurements were taken 5 times between the unchanged version and the one with changed listed above. Figure 4.2 shows the means of the *fetchSubscription()* method for each of the 5 repetitions. The mean was calculated of all taken measurements of the method during the run. This is one of the changed methods and it was expected to be generally slower in response times. The means of the two versions are fluctuating around the same numbers. The same is shown in Figure 4.3 for the method *subscribe()*. Here the first repetition seems to vary highly, while the next 4 seem to be around the same value. This is also a changed method. The means for a third method can be found in Figure 4.4. This time they are for the *unsubscribe()* method. This method was not touched, but has similar fluctuations to the other two methods of the same class.

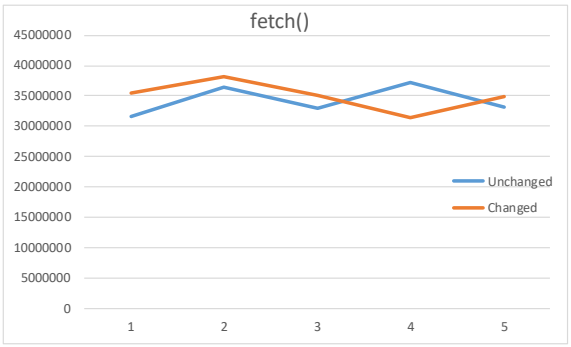


**Figure 4.2.:** Means over a whole measurement run for *fetchSubscriptions()* method.

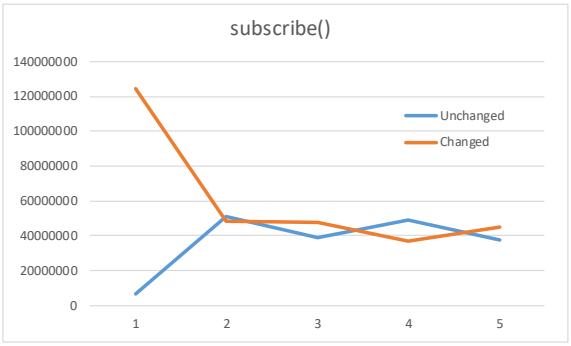


**Figure 4.3.:** Means over a whole measurement run for *subscribe()* method.

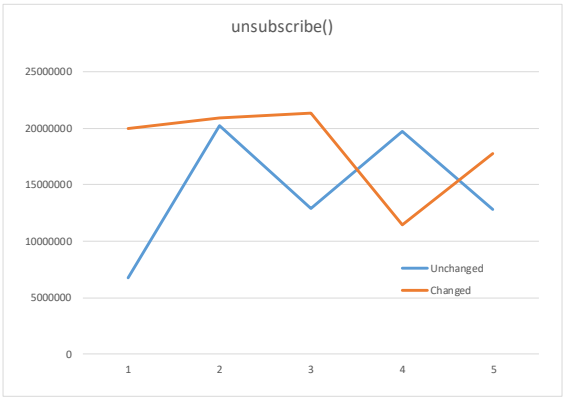


**Figure 4.4.:** Means over a whole measurement run for *unsubscribe()* method.

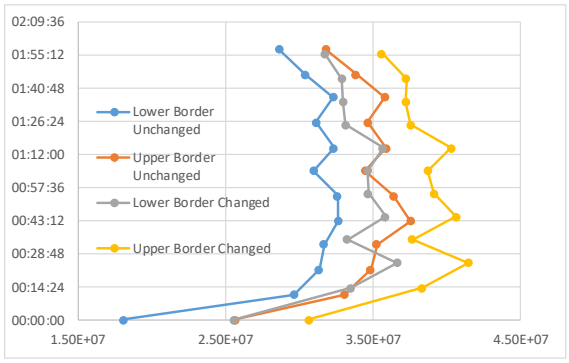


**Figure 4.5.:** Confidence intervals over 2 hours for the *fetchSubscriptions()* method.

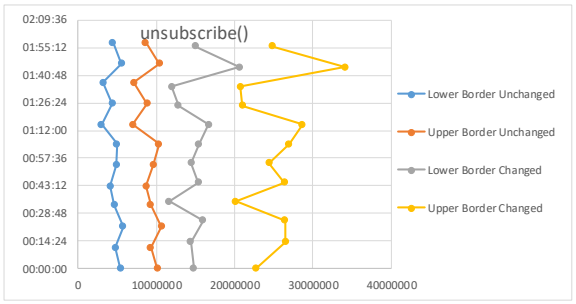


**Figure 4.6.:** Confidence intervals over 2 hours for the *unsubscribe()* method.

Another factor, that has to be mentioned, is that the number of taken measurements per run fluctuated also. In around half the cases the between 125,000 and 150,000 measurements were taken. During the remaining measurements about 275,000 to 300,000 response times were recorded. This didn't seem to impact the mean of the response times though. The standard deviation during the measurements was around 0.8 to 1.2 times the mean, which is pretty high.
Figure 4.5 shows the confidence borders for the *fetchSubscriptions()* method during the two hours of the first measurement. Here it can be seen, that they overlap quite often, in other words they are evaluated as not significantly different, even though changes were made to method to reduce its performance. Figure 4.6 shows the same graph for the unchanged method *unsuscribe()*. Here the opposite is available and there is a significant difference. Every interval would be detected as a performance regression.

As for Git Bisect, since regressions were detected in any commit and method, doing GIT Bisect was impossible. Since results varied vastly between the measured commits, commits where no changes happened were either improvements or regressions compared to the initial commit. The call tree analysis behaved similar to this, since sometimes in-between methods would be flagged, sometimes they weren't.

## 4.4. Discussion

In the following the approach will be discussed in regards to the research questions, which were stated above.

### 4.4.1. RQ1: Reliability

The reliability as observed during evaluation falls with the observed results. The load tests conducted seem to be an unreliable source of performance data. One reason for this may be how the load tests were executed. There may have been too many outside factors, that changed the measurements. An more isolated environment may have led to more stable measurements. One point may be, that Locust and the RSS-Reader were both in the same Kubernetes cluster.
Another problem could lie with the proof-of-concept implementation, that was only able to take a part of the measurements at once. While around 400 measurements for a method may seem as a good basis for computing the confidence intervals, compared to Heger et al.'s 50 measurements, it may simply not be enough in order to filter out random spikes in response time.
As a result of the problems with regression detection, GIT-Bisect does not work correctly and can't be used.

### 4.4.2. RQ2: Scalability

The approach itself in theory seems to scale well. The load test length can be altered and the number of measurements either lowered or raised. As far as the implementation goes it didn't run into any problems with scalability yet. The time needed goes up with number of measurements. One point, where problems may appear, is, if the number of measurements gets too high and the computation of confidence intervals runs into problems because of the maximal possible long value.

### 4.4.3. RQ3: Improvements

One improvement was already mentioned in a previous section: The computation of confidence intervals should be improved to be able to take a whole set of data, not only a part. In the evaluation this may have made a difference when computing the confidence intervals. Another improvement, that could be done would be to prepare the data for confidence interval computation, by for example removing vast outliers.

In the area of call tree analysis one improvement that could be made for the implementation would be to count the number of times a certain call tree appears. This number of appearances may have influence when a method comes with optional methods. If a certain call-tree with a long running contained method becomes more frequent this could lead to a regression being detected, even if no change was made. By counting this case could be removed and the number of false-positives would be lowered.

### 4.4.4. RQ4: Call Tree Analysis before GIT-Bisect

The idea behind this was to do a call tree analysis before doing GIT-Bisect in order to identify classes and methods, which are part of the call tree of a detected regression. If GIT-Bisect selects a commit for testing, in which none of the classes are touched, that could influence the as regression marked method, one could simply use the GIT-Bisect skip function, where another new commit is chosen for testing. By doing this only commits, that could've influenced the identified method are tested. In a setup similar to the evaluation setup, this could potentially save much time, since tests were run for multiple hours. If one wants to make sure, that the correct commit is identified, the last marked 'good' commit and the commit identified by Bisect could be used to do a second GIT-Bisect to find the exact commit. A slight alteration to this is to do the test on the commit right before the identified commit of the first Bisection, since then it can be learned, whether the error results from the identified or another commit.

### 4.4.5. Threats to Validity

There are external and internal threats to validity, as this was an experiment.

#### Internal

During measurements the test setup was influenced by unknown factors. This is visible in form of strong fluctuations of the mean of the measurements. These factors are could be reduced by further isolating the test environment in order to reduce possible influencing factors. Even though the number of measurements per method were in most cases around 2,000 and above, the time the tests are run could be raised further, in order to get more data. Also by doing measurements, like in case of the experiment using Kieker, the recorded measurements are influenced. A last point is the implementation of the proof-of-concept implementation, that lets it only analyze 25,000 measurements at a time may have influenced it.

External

The RSS-Reader is a relatively small application, with small call-trees. Industrial systems may be much bigger and even though the approach should scale well, it wasn't tested with bigger systems. Another point is, that the server, the experiments were done on, may have varying performance depending on the time of day due to more or less usage. This may also have influenced the measurements.

Chapter 5

# Conclusion

During this work a method to test for performance regressions using load testing and searching for their root cause was introduced. The approach used data, that was gathered during load testing with existing load tests of a project, in order evaluate the performance on one commit. This was then used to compare with the performance of earlier versions in order to find performance regressions using confidence intervals over the response time mean. Next possible root causes were identified by utilizing GIT's GIT-Bisect function and a call tree analysis. This second step is done in order to help the developer fix the problem faster and easier. The approach can be used in continuous integration environments, where load tests exist and are run regularly.
This approach was implemented in form of a proof-of-concept implementation, which was evaluated by using test data generated using the Netflix RSS-Reader [net14]. For the evaluation Regressions were introduced and the implementation was used to discover them.

Overall the evaluation showed, that there are flaws in the approach when the performance data varies too much. However the approach may work, when the data is taken from a more isolated area and as such is less influenced. If the problem with the input data can be resolved, the approach may be a viable addition to the field of performance regression testing. The future work states some factors, that may improve the approach.

## Future Work

One part of the future work is to evaluate the approach fully automated. This includes using a fully automated testing setup as opposed to the one used during evaluation, where tests were manually started. An example for a tool-setup that could be fully

automated is presented in Appendix A.
Another part, that has yet to be automated is the GIT-Bisect usage within the proof-of-concept implementation. Therefore, the testing setup should also be automatically controlled from the implementation.
The evaluation should further be extended by using a real-world software as the test program, where a regression occurred at an earlier time. This would be a more valid approach to evaluate the implementation and the approach.
A fourth and last point would be to simply evaluate further. Scalability of the approach was not evaluated as much as it was planned, and could be looked into more. And the flaw for confidence intervals in the proof-of-concept implementation should be removed and retested with an value over all measurements.
The call tree analysis part of the root-cause isolation could be further extended and enhanced. An example for this would be the improvement proposed during evaluation.

Appendix A

# Extended Toolchain for automation

In the following a toolchain will be described, that may be fully automated. For this the tools not used in the thesis will be presented, followed by a short overview how the final tool chain may look like.

## A.1. Gatling/JMeter

Gatling[1] and JMeter[2] are tools that can be used in order to put load on a system. This allows to do load tests on a system, that is to be evaluated. Creating these load tests early and doing them, will enable the developer to detect bottlenecks and performance problems early, reducing the time needed to test later. This is also enhanced by the fact, that it is possible to script automatable tests, which simulate multiple users with different actions. It is possible to integrate both tools into a continuous delivery pipeline.

## A.2. Jenkins

Jenkins[3] is an open source automation server for software development. It is possible to use it in order to automate steps like building, testing and deployment. It is very extensible, since it supports the use of plugins, which can be added into a 'pipeline' in order to build/test/deploy the software in the desired way. There is a multitude of existing plugins that can also be downloaded and used. Jenkins itself can be run either

[1] http://gatling.io/
[2] http://jmeter.apache.org/
[3] https://jenkins.io/doc/

by using native system packages, installing it and running it as a Java-application or by using a Docker-Container. This makes it very portable and system-independent.

The earlier mentioned ‘pipeline’ is used in order to automatically guide a project from source code to a deployed version. This pipeline first has to be configured by the user. After configuring it, Jenkins automatically takes the code (either from the system’s own memory or an online version-control platform such as Git-Hub) and builds the project. After a successful build is available it can be either tested or deployed (or both) automatically.

## A.3. Toolchain overview

As a base for the tool chain serves an OpenStack Cloud, where a Kubernetes instance runs on top. In Kubernetes a Docker-Container with Jenkins will be running. Jenkins will pull a test-project from GitHub, build it and deploy it containerized into the Kubernetes cluster, using a plugin to manage the container. Gatling or JMeter can then be used in order to run load tests on the application then and Kieker enables the measurement of performance data from the system under load. This results in a toolchain, that is able to automatically pull, build, deploy and test a repository from GIT.

Appendix A

# Bibliography


[AHR02] M. Arnold, M. Hind, B. G. Ryder. “Online feedback-directed optimization of Java.” In: *ACM SIGPLAN Notices*. Vol. 37. 11. ACM. 2002, pp. 111–129 (cit. on pp. 9, 10).

[BA07] U. Bellur, A. Agrawal. “Root cause isolation for self healing in J2EE environments.” In: *Self-Adaptive and Self-Organizing Systems, 2007. SASO'07. First International Conference on*. IEEE. 2007, pp. 324–327 (cit. on p. 12).

[BBH+17] L. Bulej, T. Bureš, V. Horky, J. Kotrč, L. Marek, T. Trojánek, P. Tuma. “Unit testing performance with stochastic performance logic.” In: *Automated Software Engineering* 24.1 (2017), pp. 139–187 (cit. on p. 8).

[BC08] M. Bayan, J. W. Cangussu. “Automatic feedback, control-based, stress and load testing.” In: *Proceedings of the 2008 ACM symposium on Applied computing*. ACM. 2008, pp. 661–666 (cit. on p. 7).

[BD99] G. Banga, P. Druschel. “Measuring the capacity of a Web server under realistic loads.” In: *World Wide Web* 2.1-2 (1999), pp. 69–83 (cit. on p. 7).

[BGK+02] O. Ben-Yitzhak, I. Goft, E. K. Kolodner, K. Kuiper, V. Leikehman. “An algorithm for parallel incremental compaction.” In: *ACM SIGPLAN Notices*. Vol. 38. 2 supplement. ACM. 2002, pp. 100–105 (cit. on p. 9).

[BKT04] L. Bulej, T. Kalibera, P. Tuma. “Regression benchmarking with simple middleware benchmarks.” In: *Performance, Computing, and Communications, 2004 IEEE International Conference on*. IEEE. 2004, pp. 771–776 (cit. on p. 10).

[BM03] S. M. Blackburn, K. S. McKinley. “Ulterior reference counting: Fast garbage collection without a long wait.” In: *ACM SIGPLAN Notices*. Vol. 38. 11. ACM. 2003, pp. 344–358 (cit. on p. 9).

[BOP03] K. Barabash, Y. Ossia, E. Petrank. “Mostly concurrent garbage collection revisited.” In: *ACM SIGPLAN Notices*. Vol. 38. 11. ACM. 2003, pp. 255–268 (cit. on p. 9).

[Car17] Carlos Palminha. *Using "Continuous Integration" Practices for SoC Developmen*. 2017. URL: https://www.synopsys.com/company/resources/newsletters/prototyping-newsletter/continuous-integration-practices.html (cit. on p. 18).

[Cor08] S. P. E. Corporation. *SPEC JVM98 Benchmarks*. 2008. URL: http://www.spec.org/jvm98/ (cit. on p. 9).

[Cou08] C. Couder. “Fighting regressions with git bisect.” In: *Online: https://www.kernel. org/pub/software/scm/git/doc s/git-bisect-lk2009. html, The Linux Kernel Archives, GIT Repository, Version* 4.5 (2008) (cit. on p. 22).

[CS14] S. Chacon, B. Straub. *Pro git*. Apress, 2014 (cit. on p. 22).

[DCBK15] A. Danciu, A. Chrusciel, A. Brunnert, H. Krcmar. “Performance awareness in Java EE development environments.” In: *European Workshop on Performance Engineering*. Springer. 2015, pp. 146–160 (cit. on p. 12).

[DGH+06] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, G. Weber. “Realistic load testing of web applications.” In: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. IEEE. 2006, 11–pp (cit. on p. 7).

[GBE07] A. Georges, D. Buytaert, L. Eeckhout. “Statistically rigorous java performance evaluation.” In: *ACM SIGPLAN Notices* 42.10 (2007), pp. 57–76 (cit. on pp. 9, 11, 20, 22).

[Ghe05] G. Gheorghiu. *Performance vs. load vs. stress testing*. 2005. URL: http://agiletesting.blogspot.de/2005/02/performance-vs-load-vs-stress-testing.html (cit. on p. 6).

[GIT17] GIT. *Git-Bisect v2.13.0 Documentation*. 2017. URL: https://git-scm.com/docs/git-bisect (cit. on pp. 11, 22).

[Gre06] Greg Linden. *Marissa Mayer at Web 2.0*. 2006. URL: http://glinden.blogspot.de/2006/11/marissa-mayer-at-web-20.html (cit. on p. 1).

[HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010 (cit. on p. 13).

[HHF13] C. Heger, J. Happe, R. Farahbod. “Automated root cause isolation of performance regressions during software development.” In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM. 2013, pp. 27–38 (cit. on pp. 1, 2, 8, 9, 11, 17, 20).

[HLM+15] V. Horky, P. Libič, L. Marek, A. Steinhauser, P. Tuma. “Utilizing performance unit tests to increase performance awareness.” In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM. 2015, pp. 289–300 (cit. on pp. 8, 12).

[HRN06] J. Humble, C. Read, D. North. “The deployment production line.” In: *Agile Conference, 2006*. IEEE. 2006, 6–pp (cit. on p. 13).

[JD 07] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Chapter 2 – Types of Performance Testing*. 2007. URL: https://msdn.microsoft.com/en-us/library/bb924357.aspx (cit. on p. 6).

[JH15] Z. M. Jiang, A. E. Hassan. “A survey on load testing of large-scale software systems.” In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118 (cit. on pp. 2, 7, 19).

[KBT05] T. Kalibera, L. Bulej, P. Tuma. “Automated detection of performance regressions: The Mono experience.” In: *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*. IEEE. 2005, pp. 183–190 (cit. on p. 10).

[Kie17] Kieker research group. *Kieker keeps an eye on your software*. 2017. URL: http://kieker-monitoring.net/ (cit. on p. 33).

[KIZ+14] T. Kalbarczyk, N. Imam, T. Zhang, Y. Gotimukul, B. Boyles, S. Patel. “Analyzing Performance Differences between Multiple Code Versions.” In: (2014) (cit. on p. 12).

[KT06] T. Kalibera, P. Tuma. “Precise regression benchmarking with random effects: Improving Mono benchmark results.” In: *European Performance Engineering Workshop*. Springer. 2006, pp. 63–77 (cit. on p. 11).

[Men02] D. A. Menascé. “Load testing of web sites.” In: *IEEE Internet Computing* 6.4 (2002), pp. 70–74 (cit. on p. 7).

[MH06] P. McGachey, A. L. Hosking. “Reducing generational copy reserve overhead with fallback compaction.” In: *Proceedings of the 5th international symposium on Memory management*. ACM. 2006, pp. 17–28 (cit. on p. 10).

[MSB11] G. J. Myers, C. Sandler, T. Badgett. *The art of software testing*. John Wiley & Sons, 2011 (cit. on p. 5).

[Nat08] Nati Shalom. *Amazon found every 100ms of latency cost them 1% in sales.* 2008. URL: https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (cit. on p. 1).

[net14] netflix-oss-recipe group. *NetFlix-RSS Reader*. 2014. URL: https://github.com/Netflix/recipes-rss (cit. on pp. 32, 41).

[NTY07] D. Nir, S. Tyszberowicz, A. Yehudai. “Locating regression bugs.” In: *Haifa Verification Conference*. Springer. 2007, pp. 218–234 (cit. on p. 9).

[OAB12] H. H. Olsson, H. Alahyari, J. Bosch. “Climbing the" Stairway to Heaven"–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software.” In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE. 2012, pp. 392–399 (cit. on p. 13).

[OHH+16] D. Okanović, A. van Hoorn, C. Heger, A. Wert, S. Siegl. “Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces.” In: *European Workshop on Performance Engineering*. Springer. 2016, pp. 94–108 (cit. on pp. 26, 28).

[Pat01] R. Patton. *Software testing*. Sams publishing, 2001 (cit. on pp. 5, 6).

[RHWP15] A. A. U. Rahman, E. Helms, L. Williams, C. Parnin. “Synthesizing continuous deployment practices used in software development.” In: *Agile Conference (AGILE), 2015*. IEEE. 2015, pp. 1–10 (cit. on p. 13).

[VGG+13] A. Vulimiri, P. Godfrey, S. V. Gorge, Z. Liu, S. Shenker. “A cost-benefit analysis of low latency via added utilization.” In: *arXiv preprint arXiv:1306.3534* (2013) (cit. on p. 1).

[VWH12] A. Van Hoorn, J. Waller, W. Hasselbring. “Kieker: A framework for application performance monitoring and dynamic software analysis.” In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, pp. 247–248 (cit. on pp. 14, 28).

[Wey98] E. J. Weyuker. “Testing component-based software: A cautionary tale.” In: *IEEE software* 15.5 (1998), pp. 54–59 (cit. on p. 6).

[ZAH12] S. Zaman, B. Adams, A. E. Hassan. “A qualitative study on performance bugs.” In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE. 2012, pp. 199–208 (cit. on p. 9).

All links were last followed on June 12, 2017.

**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