MASTER'S THESIS 2022

How to Improve Feedback and Traceability for Performance in Software Development

Hanna Höjbert, Elias Vernersson

ISSN 1650-2884
LU-CS-EX: 2023-79

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-79

How to Improve Feedback and Traceability for Performance in Software Development

Hanna Höjbert, Elias Vernersson

How to Improve Feedback and Traceability for Performance in Software Development

Hanna Höjbert
ha0223ho-s@student.lu.se

Elias Vernersson
el2121ve-s@student.lu.se

May 19, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Anton Nilsson, anton.nilsson@telavox.se

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Identifying and managing performance issues is a challenging task that often needs a defined practice in the development process. Without a practice, there is a chance that performance issues are pushed into production and thus affect the end user. Making sure an application is performing well is, however, not trivial and often involves relaying feedback to developers either from testing or production monitoring. Providing developers with adequate, accurate and timely feedback about the performance of an application is a considerable challenge. Today there are two approaches, monitoring the production environment and solve the issues as they are identified or testing for performance by writing performance testing suites. However, monitoring the production environment provides slow feedback and writing extensive test suites is expensive.

We therefore formulated a requirements specification with the help of developers for an automated tool. We then investigated and implemented a prototype based on the requirements that aims at providing developers with additional information about the application performance by monitoring an already existing unit testing suite.

The conclusions of the research is that the prototype could improve performance issue identification for developers before releasing into production and thereby improve the feedback loop.

Keywords: Feedback Loop, Increasing Traceability, Requirements Specification, Continuous Integration, Continuous Delivery, Performance Monitoring, Performance Issue Identification, DevOps, Closure

Acknowledgements

First of all, we would like to thank Lars Bendix for the support and knowledge he has shared with us during the master's thesis process. Throughout the work we have grown to cherish both the early mornings as well as the late Sundays evenings spent on discussing everything from how to write a better thesis to how to make the best pizza possible.

We would also like to thank our supervisor, Anton, at the case company for the warm welcome and guidance for getting in touch with the right people. We would also like to thank all developers who have been involved in various interviews and evaluations.

Contents

Chapter 1
Introduction

Without a defined practice in the development process for how performance issues can be identified, there is a good chance that they will be pushed into production and to the end users. For this thesis, the case company, Telavox, ran the risk of this particular problem of performance issues being released into production as they had no proper feedback system implemented, in terms of finding and solving performance issues in the code - a problem that, in the long run, could lead to degraded user experience. User experience is a central part of what makes a service successful with one of the deciding qualities being responsiveness. Performance issues that did occur were at best caught during the production stage, after which data about performance was manually retrieved and analyzed by developers. This took between a few hours and about a week depending on how severe the issue was, with a lot of time spent on identifying where in the code the problem was and who had introduced it. The lack of traceability was the main culprit, once a problem had occurred it was hard for the developers to identify what part of the code had caused the issue and in which version of the code it was introduced.

There are several reasons why this is an important problem. The main reason why is the costly and time-consuming resources that the company needs to put into identifying performance issues that instead could be put into developing and improving the product. Another reason is that performance issues create a frustrating user experience in which the user has to wait for slow requests to the system. Keeping the performance of a system sufficient is important in order for a company to retain its customers while also minimizing the risk of deterring new potential customer, which could have an impact on the company's revenue.

To solve the problem above this thesis aims to improve feedback and traceability in the software development process. This thesis contributes to current research within three fields; feedback, traceability and closure. This was achieved by diverging from current standard methods, namely performance testing, and instead rely on already existing unit tests as a basis for testing performance. The first contribution is the method of increasing traceability in the system. unit tests are combined with performance monitoring in order to create rudimentary performance tests in the build stage, with little time and effort needed to write extensive

test suites. Furthermore, the research has contributed by investigating what requirements a developer has on a tool for detecting and resolving performance issues found in the code. Lastly, it was investigated how the sense of closure was affected by extending the testing stage with additional, albeit experimental, performance testing. These conclusions were ultimately found by first asking the four research questions below.

RQ1 What effect could an automated tool for performance have on the feedback loop and development process?

RQ2 What is useful information for the developer, what requirements does the developers have on the automated tool for performance?

RQ3 How would the time required for bug fixing be affected with the use of an automated tool for performance?

RQ4 How could a shorter and rigorous feedback loop give developers faster closure?

Several steps were performed to investigate the research questions. An overview of the methodology can be seen in figure 1.1. Two rounds of interviews were conducted during the project, one in the first step of project, which had the purpose to explore the problem domain and get an insight into the development process. In the next step a second round of interviews were conducted in order to elicit requirements that a developer would have on a tool. The first step in which two different interviews were conducted with developers at the case company to get an understanding of the problem, get an insight in the work process and derive a requirements specification for the tool. Furthermore, a questionnaire was sent out to receive answers to set a baseline for the current situation. The next step was to dive into literature to lay an academic foundation and find similar experiments to this thesis. Then, several prototypes were designed and explored where one was chosen to be implemented. In the last step, the prototype was tested and evaluated by developers in a testing session with a following interview.

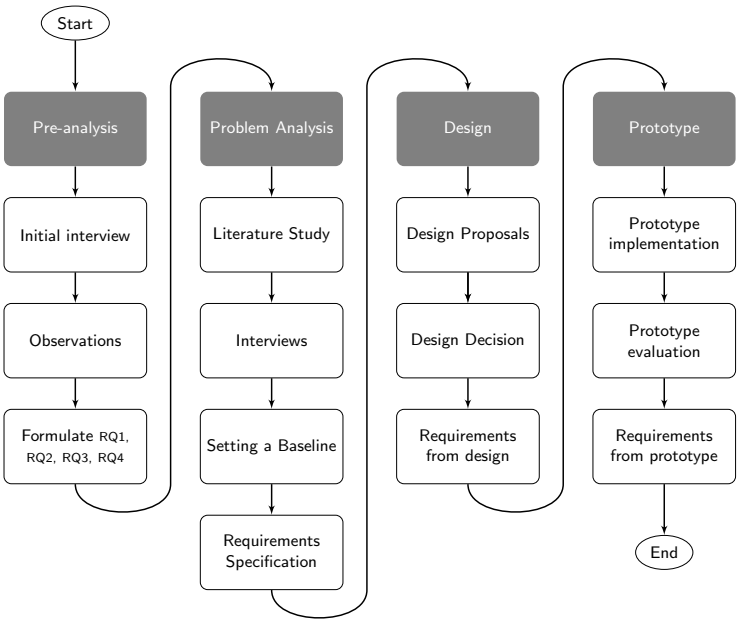

Figure 1.1: The figure shows the methodology used in the project.

The report is composed of a background chapter which starts with a pre-analysis of the problem domain, followed by a theoretical framework on which the thesis is based upon and a motivation of the methodology for the thesis. Moving on, the problem analysis chapter which will explore similar approaches from articles, motivate the questionnaires and interviews, while also relating preliminary data to form an initial requirement specification. The design chapter will then motivate how a prototype was chosen, implemented and evaluated, along with motivating additional requirements that were elicited during the design phase. In the results chapter the results from the questionnaire, interviews, the requirements specification itself and the prototype will be discussed along with deriving core requirements, followed by a chapter in which the research along with related and future work will be discussed. The research will then be summarized in the conclusion of the report.

Chapter 2
Background

In this chapter we will relate and analyze the case company to the context of the thesis in order to formulate research questions from the initiating problem, discuss related theory that is relevant to the initiating problem and motivate the methodology as an extension of the research questions. This is important in order for the reader to understand how the context has influenced decisions and approaches used in the thesis, but will also help the reader get familiar with the case company. It also helps the reader understand the initiating problem on a more granular level by relating the initiating problem and context to relevant theory, while also serving the reader as a source of information that can be consulted when reading the report if needed. First the context at the company will be analyzed in order to develop relevant research questions, that act as a proxy to solving the initiating problem. The context of the thesis will then be related to relevant theory. Lastly, a methodology will be established from analyzing the research questions and investigating possible approaches.

2.1 Problem Understanding

The purpose of this section is to explore and expand the initiating problem as well as investigate research questions that encapsulate the essence of the initiating problem. This will be done by first investigating the case company, then conducting a pre-analysis to find a basis that will lastly be used to develop research questions for the thesis.

2.1.1 Telavox

Telavox, which will be addressed as the case company in this thesis, the provides a cloud based uCaaS, unified communication as a service, to make it easier for businesses to manage their communications. They provide telephony, PBX, messaging, meetings and contact center all in one platform. The guiding star for development of the product is simplicity and the company owns their platform from top to bottom which means that they can steer the product

in the direction they want and be adaptable to what customers want. With the product, it is easy for businesses to scale their workers up or down as the admin can self-administer the set up of the product in the admin portal. Today, the company has 300 000+ users across 60 000 customers and are located in 9 countries around the world. Their headquarters are located in Malmö where the main development of the product takes place with around 90 developers working there. There are many departments at the company where one of them is DevOps which is where this thesis will be conducted.

2.1.2 Pre-analysis

In order to develop relevant research questions for the initiating problem there was a need to investigate the current situation at the case company. This was done by conducting interviews with two developers that had experience with performance issues at the case company. Furthermore, observations made by us will be explored. First off, the interviews and its findings will be investigated, then the observations will be explored.

Initial Interviews

From the interviews it could be derived that developers consider the main problem, in terms of performance issues, being that requests to the database take too long. This locks the database for other users which in turn results in a degraded user experience. This originates in that developers might not have a complete understanding of how their code affects the database performance.

There seemed to be no single way for developers of testing the performance of the service, but instead every developer had their own method for making sure their code is good enough in terms of performance. Some developers tested their code manually by clicking around in the application to see if something was noticeably slow. Other developers used a profiling tool that allowed them to see what parts of the code took time when performing manual testing. A third method was to ask the peer reviewer of the code to also check if the code was good enough in terms of performance. A shared commonality between the methods was that it was up to an individual to decide whether or not the code is good enough.

It was also mentioned that developers might be surprised to know their code is slow since there were no prior warnings and therefore they assumed it was good enough. But it could also be surprising due to the fact that the system might not behave as the developer had thought it would. This thought of always having to make sure the code is fast enough might cause the developer to be overly cautious in committing changes and might also be seen as an extra burden to the developer.

The interviewees described that there is a specially appointed group of developers that, in addition to their regular work, are tasked with monitoring and solving performance issues that are identified in production. The slowest performance issues that had occurred in the production environment were then logged. Once a week the group then chose issues to resolve, based on their importance and in some occasions fixed issues within a few hours if it was an urgent issue that affected large parts of the system. The issues were either resolved by the group or dispatched to the developer responsible for introducing the issue. Figuring out what part of the code caused the performance issue was not always easy as the logged query was generated dynamically and therefore was hard to read, which resulted in it being

time consuming to find the responsible developer. In case the the developer responsible for the issue could not be found, the issue would be manually added into JIRA, the project management system at the case company, or dispatched to another suitable developer or team. The interviewees pointed out that it thus normally takes about a week to get feedback if their code had caused any performance issues. Furthermore, they describe that this will cause a gap in time between coding and handling any performance issues, which means that developers lose context and workflow.

On the question if the case company uses continuous delivery or continuous deployment both interviewees answered that the company uses Continuous Delivery.

To summarize, the method testing of performance differ from developer to developer, with some testing manually by clicking around in the application and other by using profiling tools. Furthermore, there exists a feedback loop from production, regarding the performance of the system, that often takes at least a week from which developers will be notified in different ways.

Observations

After having conducted interviews there was still a need to identify current used practices at the case company, that might have been glanced over in the interviews, such that this context could be taken into account when developing the research questions. The development process was therefore observed and the observations will be lined out from the beginning of the development process to the end.

In the development stage, the case company encourage developers to adopt test driven development (TDD). To test performance of requests to the database, developers used a separate program to try a query out against a test database and decided themselves if performance was good enough.

As mentioned in interviews and from further investigation, we found that the case company implements continuous integration (CI) and continuous delivery (CD). In the CI stage, the case company applies git as their version control software and the shared code base is stored on GitHub. A new git branch is made every time a developer wants to implement changes in the code base. When changes are made, the developer creates a pull request to the main branch. This triggers the build server to build the code and perform more thorough testing but we observed that performance testing is still missing in the test suite.

In the CD stage, a new build is deployed manually to production by developers a few times a day and each build contains several commits with new changes from several developers. In production, the case company monitors all requests coming to the system to make various analyzes.

2.1.3 Research Questions

In the project there is a need for guidelines in order to make sure the research does not diverge from the initiating problem. Therefore a number of research questions have been formulated and by answering the research questions, the initiating problem can also be solved. This subsection will thus reflect on the previous subsections in order to identify needs that are determining factors to solve the initiating problem. The research questions are prioritized

in order of significance to the initiating problem. We then discuss the formulation of the research questions in this order.

From the interviews and observations, several problem areas were identified. The first of which was that the feedback loop took at least one week as well as it being performed manually. The second problem was that the traceability between a performance issue in production and the code was poor.

RQ1 was formulated with the intention that it would be interesting to investigate whether or not an automated tool for performance measurements could possibly solve these issues and how it would affect the feedback loop as well as the development process as a whole. The reason why this is interesting is because it has the possibility of reducing the amount of work needed to identify and relay feedback about performance issues. It is also interesting as an automated tool would fit into the current workflow since the case company use DevOps practices.

Since this thesis investigates whether or not an automated tool would have any effect on the development process, it is also of interest to investigate what requirements developers would have on such a tool. In order to make sure a tool contributes with the highest amount of satisfaction to the developers, it is necessary to investigate what requirements they have on such a tool. A requirement specification could then act as a good basis for future research or development as it would give an indication of where to start a similar project. In order to investigate this **RQ2** was formulated.

As mentioned earlier it was interesting to know what effect an automated tool could have on the problem areas stated earlier. During the interviews it was found that identifying the location in the code that caused a performance issue was both difficult and time consuming. It was therefore interesting to see if an automated tool could have any impact on the amount of time required to fix bugs. This is interesting because an automated tool could improve the traceability and thereby make bug fixing easier. **RQ3** was therefore formulated with the intention of investigating this.

Lastly, it was mentioned that there is a gap between the time a developer writes code and the time she or he receives feedback. It was also said that not knowing whether or not you have tested your code enough was perceived as an extra burden to the developer, which could be a result of a lacking testing stage. This could contribute to developers not feeling closure early in the development process. It was therefore interesting to investigate if developers could experience faster closure by receiving faster feedback from an automated tool. **RQ4** was formulated in order to investigate this further.

RQ1 What effect could an automated tool have on the feedback loop and development process?

RQ2 What is useful information for the developer, what requirements does the developers have on the tool?

RQ3 How would the time required for bug fixing be affected with the use of an automated tool?

RQ4 How could a shorter and rigorous feedback loop give developers faster closure?

2.2 Theoretical Framework

This section will use the research questions developed in the previous section, and relate it to theory. This is essential in order for the reader to have an adequate understanding of the theory as it will function as a basis for the rest of the report, as well as a source of information. First we will explore some theory that is related to the research questions and then investigate some theory that is brought up later in the report.

2.2.1 Traceability

This section will focus on the theory behind traceability, where traceability can be found in the software development process and the value it adds to the process. Also, the theory will be related to the research questions.

Good traceability allows for a configuration item to be traced throughout the entire software development life cycle. Anything that is used or created throughout the software development life cycle, referred to as an artifact, can be turned into a configuration item [3]. Traceability reveal itself not only as a timeline of changes made to a single configuration item, such as code commits, but also as a relationship between a feature and the requirement that generated the feature as well as the tests cases for the feature. These relationships is what allows for a configuration item to be followed throughout the software development life cycle. Testing code exemplifies traceability well. Good traceability for example allows the developer to see what tests passed and what tests failed and are given the information about the version of the tested system. The developer can thereby observe the version of the system that was tested and see what changes has been made since the last time the tests passed. This information could then be used in order to deduct what has caused the tests to fail. Without traceability, it would be hard to know what version was tested and what changes had been made since the last time the tests all passed. This relates to the research questions as it is the lack of traceability between logged performance issues and the code itself that makes it difficult to deduct what part of the code caused the performance issue. An increased traceability between this artifact and the code could make it easier for the developer to find the code that caused the performance issue.

2.2.2 Feedback

In order to fully understand the research questions it is necessary to investigate the theory behind feedback and its importance in the software development process. As part of feedback, Test Driven Development will be highlighted and explored to motivate the advantages of implementing it.

Krancher et al. refers to feedback as ”information about actions returned to the source of the actions”, which is a definition that suites this thesis. Furthermore, Krancher et al. mentions that feedback is received, for example, when a person testing the code informs the developer about results but also when compiling, testing or reviewing code [11]. Feedback allows developers to see the results of their actions and act accordingly. M. Poppendieck and T. Poppendieck mentions ”In most cases, increasing feedback, not decreasing it, is the single most effective way to deal with troubled software development projects and environments” [14].

M. Poppendieck and T. Poppendieck furthermore describes that, generally, dynamic environments require more feedback as opposed to static environments. It could therefore be argued that software development generally would benefit from more rapid feedback. Test driven development is from this standpoint a way of increasing fast feedback by continuously testing whether an implementation achieves the functionality that it intends to. Rather than implementing functionality and receiving feedback days later from a test group, the feedback can be received within a matter of minutes rather than days.

2.2.3 Automation

Automation is the act of making manual processes automated in order to save time and resources while also saving developers from having to carry out tedious and repetitive tasks. Automation has the benefit of being consistent, fast and less resource intensive compared to carrying out tasks manually. Testing is a task that is often subject to automation. M. Poppendieck and T. Poppendieck mentions "The most effective way to facilitate change is to have an automated test suite that tests the mechanisms the developers intend to implement and the behavior the customers need to have" [14], which is a compelling reason to automate it. But automation can also be applied to a broader set of practices, rather than just testing.

2.2.4 DevOps

In order to understand the context of the thesis it is important to have a brief knowledge of DevOps. It is also important as the concept will be referred to throughout the report.

According to Bass et al. "DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" [1]. What practices actually constitute DevOps is debated, but Zhu et al. mentions that a DevOps developer is able to place their code into production without need for coordinating with other development teams and after the deployment the system should be monitored [17]. DevOps developers are, in that sense, cross-functional. Furthermore Zhu et al. mentions that DevOps heavily rely on tools to perform the processes between committing a change and deploying into production. Ebert et al. state that "Quality deliveries with short cycle time need a high degree of automation" [6], which could be the reason for their heavy reliance on tools.

2.2.5 Continuous Integration and Delivery

This subsection will briefly discuss the theory of continuous integration and delivery. This is important as the case company implements continuous integration and delivery in their development and as such the research questions will be investigated with this context.

Continuous integration and delivery is the combination of two different practices: continuous integration and continuous delivery. Continuous integration is the act of frequently integrating new code to the main branch and then automatically testing the integrated code. Continuous delivery, on the other hand, is an extension of continuous integration that also automatically puts the system in a state that it can be deployed into production by the press

of a button. Chen explores the benefits and challenges of implementing continuous delivery [5]. Chen mentions six benefits, accelerated time to market, building the right product, improved productivity and efficiency, reliable releases, improved quality, and improved customer satisfactory.

2.2.6 Test Driven Development

This subsection aims at exploring relevant theory behind the concept test driven development. This is relevant to the research questions with the reason being that test driven development is practiced at the case company and that it is important to take this into consideration.

Beck describes the test driven development as a process were the developer first writes a test that does not work, then makes the test work quickly, and lastly refactors the code [2]. The reason for refactoring the code is, according to Beck, to eliminate all of the duplicate and bad code that was introduced when just trying to make the test work. Beck also explains that an advantage with test driven development is how it shortens the feedback loop on design decisions. Likewise, M. Poppendieck and T. Poppendieck describes that tests allow developers to get immediate feedback on whether or not their implications and code works [14]. M. Poppendieck and T. Poppendieck also mentions that tests make it safe for developers to try things that otherwise would be too dangerous, since they have the tests as a safeguard.

2.2.7 Performance Testing

Monitoring the performance of a system is a reactive way of identifying performance issues. Testing for performance is, however, a proactive approach to identifying performance issues. Performance testing does not relate to the research questions directly, but will rather be investigated as a method of answering the research questions. As such a brief understanding of the concept is needed. Performance is an integral part of any service as it dictates how well a user is able to use the service in question. Or as Ian Molyneaux puts it, "A well-performing application is one that lets the end user carry out a given task without undue perceived delay or irritation. Performance is really in the eye of the beholder" [13]. It expresses the idea that performance is not necessarily about objective measurements, but rather the subjective experience of the individual user. One could argue that a benefit of performance testing is that it can be done prior to deployment, which reduces the risk of a user being exposed to a performance issue. In addition, performance testing can be automated and as such integrated into the continuous delivery process. Not only could this save time, but it would also allow each commit to be tested independently from one another, which would increase the traceability from performance to the system version. A difficulty with testing performance, however, is that the testing environment needs to be very similar to the production environment in order to produce reliable results and it is not always easy to to. This is because a production environment has live users which the testing environment somehow needs to account for.

2.2.8 Closure

Closure is an illusive concept that is referenced multiple times in the research questions. In order to be able to understand the concept and reduce the risk for ambiguity in the thesis, the concept will be defined.

Kruglanski and Webster defined the need for closure as a desire for an, or any, answer to a given topic [16]. It could therefore be argued that closure itself, is the sensation of having come to an answer on a given topic. In the context of software development and this thesis, closure will be defined as the sensation of feeling done with a feature and being able to move on to another task. This is relevant to the context of the thesis because shortening the feedback loop might have an effect on the sense of closure, since it allows the developer to know more about the current state of the system when having finished a task.

2.3 Methodology

This section will focus on discussing the use of methodology in the project, why the methodology was suitable and how it ultimately served as an extension to answering the research questions formulated earlier in this chapter. This is of interest as it reflects and motivates to the reader why the specific methodology was used and other potential approaches where abandoned, but also since it allows the reader to reflect on the methodology and recreate the research to verify our results. The section will start off by discussing the pre-analysis phase, followed by problem analysis phase, design phase and, lastly, prototype phase. An overview of the methodology of the methodology can be seen in figure 2.1.

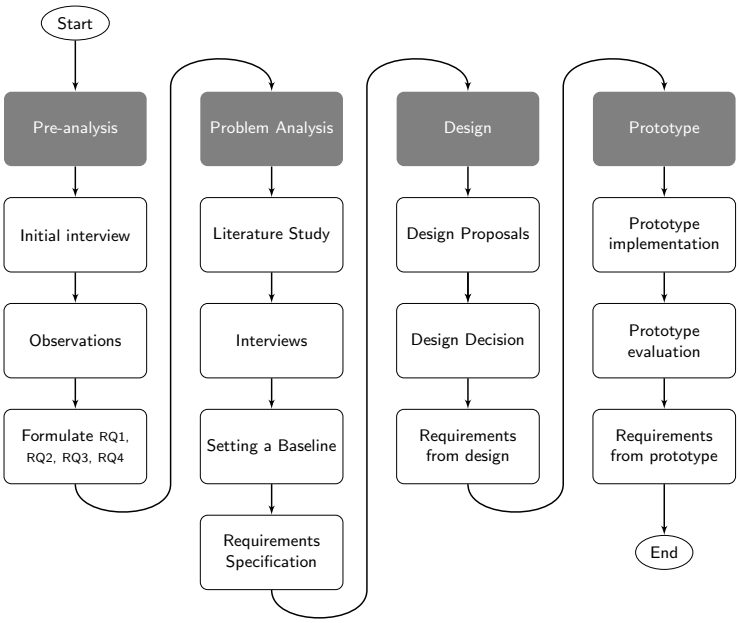

Figure 2.1: The figure shows the methodology used in the project.

2.3.1 Pre-analysis Phase

In the beginning of the research, it was essential to establish research questions that would serve as guidelines for the project and help us solve the initiating problem. This section therefore aims at motivating the approach of how the research questions were formulated through interviews with developers at the case company and observations by us of current work practices.

Initial Interview

Interviews were held with developers to get a deeper insight into the problem domain, the current development process and how the initiating problem affects the current development process. From interviews, partly, research areas were derived and research questions could be formulated.

With the help of interviews, discussions are encouraged, which means that participants discuss and sort out the ambiguities that may arise and thus create a deeper understanding of the initiating problem. It also means that new information can appear that otherwise could have been missed in e.g. a questionnaire because no further questions can be asked to the participants about what they really meant with the answer. Furthermore, the questions and answers in a questionnaire are always up for interpretation both by the respondents and conductors of the questionnaire. Therefore, new information can be thoroughly investigated by being able to ask further questions and discuss the problems.

Interviews were conducted individually so that the picture of the problem would be as nuanced as possible and developers would not influence each others answers. Two developers, who worked extensively with the problem, were asked for an interview. Both belong to the group that was started to handle and resolve performance issues in production and they have been part of the group for about 1 year.

The interview questions could be divided into four categories: what's the problem domain, how the current development process looks like regarding feedback, traceability and automation, how the problem domain affects the development process and what possible solution developers could see to the problem domain. All interview questions can be found in Appendix A. After the interviews were conducted, we realized that the area about possible solutions to the problem domain did not belong to this stage of the process. Therefore, questions regarding this area was left out in the analysis and formulations of research questions.

Observations

To complement the interviews and to create our own perception of the problem domain, we made our own observations of the development process. We explored the various parts and tools of the system and development process with the initiating problem in focus. Areas of interest for further research were found and written down for analysis together with interviews. This included searching through the repositories looking for testing frameworks and continuous integration and delivery setups.

2.3.2 Problem Analysis Phase

This subsection discuss the execution of the problem analysis phase and its constituents; questionnaire, literature study, interviews and requirements specification. Each area will be well motivated based on why the authors chose to perform each step in regards to be able to answer research questions.

Questionnaire

Choosing a method for gathering data is not always obvious and is often a balance between several different factors. In the case of this thesis it was a balance between the number of respondents, time required and accuracy. Using a questionnaire more responses could be gathered and analyzed, in comparison to conducting interviews. This is directly connected to the amount of work required to designing and sending out a questionnaire, which is considerably less compared to conducting an equal amount of interviews. Regarding the accuracy of the information gathering, it is harder to get accurate results since it is not possible to explain question or ask follow up questions like in an interview. The questions did however, at the time, seem easy enough to answer. As a counter measure against inaccurate answers, it was also tested on a developer to make sure that the questions were evident.

The reason for conducting a questionnaire was to gain a baseline of four things for the current workflow: when developers start feeling a sense of closure, the time required to identify a performance issue, the time it takes to receive feedback about performance issues and lastly to evaluate the traceability. The questionnaire included four types of questions. First, a Likert-scale which presented a statement to which the respondent could answer “strongly disagre”, “disagree”, “neutral”, “agree”, “strongly agree” or “other”. The option “other” was added in case a respondent felt none of the answers were applicable and would like to answer with a short free text. Secondly, a free text answer in which the respondents were able to write their own answers. These were used either in case the respondent would like to add anything to the previous questions, or in cases were the questions needed a more nuanced answer. Thirdly, a time estimation where the respondents were expected to answer in number of minutes, hours, days or weeks. For instance, “4 minutes” or “2 weeks”. This was used in cases when it was expected that respondents would be able to estimate.

In order to evaluate whether or not a shorter and more rigorous feedback loop has any impact on the sense of closure developers feel it is needed to have an initial baseline of measurements that the future measurements can be compared against. It was therefore of interest to measure when in the software development life cycle developers started feeling a sense of closure. Since the concept of closure is not necessarily known to all developers, it was needed to first explain the concept. The developers were then asked to estimate their sense of closure in each step of the software development life cycle, namely;

- “immediately after having written my code”,
- “when I have manually tested the code (clicking around in the application) on my local machine”,
- “when local test suits pass”,
- “when a pull request has been made”,

- "when my code has been built and tested on Jenkins",
- "when my code has been peer reviewed by two other developers",
- "when my code has been merged to main branch",
- "when my Jira issue has been resolved",
- "when my code has been deployed to production",
- "when my code has been in production for a week".

The question were answered with a Likert-scale that included five answers: "strongly agree", "agree", "neutral", "disagree" and "strongly disagree". The questionnaire aims to measure the sense of closure in every step of the software development life cycle. The answers could then give an indication to when developers on average start to feel a sense of closure. The answers were then converted to a percentage by mapping each answer to a numerical value from zero to four, in which zero represents "strongly disagree" and four represents "strongly agree". The lowest score is thus zero and the maximum possible score is four times the number of respondents. The average score of each question was therefore divided its maximum score. This has the effect that 0% means that developers on average strongly disagree with the statement, 100% means that developers on average strongly agree with the statement and 50% means that the developers on average is neutral in regards to the statement.

Literature Study

The reason behind literature study was to explore all the research questions from an academic point of view. Through the literature study, inspiration was gained from independent articles and lay a good foundation for arriving at answers for research questions.

First keywords related to the problem domain was defined and listed. Next, we independently used the set of keyword to search for articles on *Google Scholar* and *LUBsearch*. These search engines were chosen to access articles of good quality and trustworthiness. We both put the first ten articles we found for each keyword search in our respective lists. The next step was to jointly go through the lists and put the articles that appeared in both lists in a new joint list. The new list was then filtered by title and abstract to bring out the most relevant articles related to the context of research questions. After the list was reduced, we produced the final articles by reading through the introduction and summary of each article. By iteratively filtering out articles, we got time to think through an article properly as we got more and more information about it before we excluded it.

Interviews

The reason for conducting interviews was mainly to explore and gathering data for **RQ2**. Together with literature study and questionnaire, interviews served as a foundation in order to elicit requirements that the design would later build upon.

Interviews were chosen for the same reason as for background analysis as it benefits discussions and to explore the subject more deeply. Based on the literature study and questionnaire together with our own observations, questions were formulated for interviews about

what kind of feedback the developers wanted, what functions would be included, where in the development process the developer would like feedback. In addition, questions were formulated about what the development process around performance testing looked like. Furthermore, interview questions consisted of introductory questions regarding date of birth, prior education, how long the interviewees had been at the case company and what they worked with. The aim of these questions were, partly, to ease into the interview by asking the interviewees questions they could easily answer and to gather some background information for analysis as well. In hindsight some of these questions were found to be irrelevant for the research, such as their age and prior education and the interview could be eased into with some small talk prior to actually doing the interview. Still, the time an employee has worked at the case company and what their work consisted of was relevant, since it possibly could give reason to their answers. The interview questions can be found in Appendix B.

We chose to interview four developers from different teams and with different amounts of insight into the problem. We did this to get such a broad spectrum as possible because the tool was intended to be used by developers in the case company's Continuous Integration/Continuous Delivery(CI/CD) pipeline which makes developers' input on the tool momentous. Each interview was recorded on a mobile device for two reasons. The first reason was that both of us could be involved in discussions in the interviews without missing to write down anything important. The second reason was that we could go back and listen to the interviews so that we did not miss any important information.

Each interview was then transcribed into text as this would make it possible to analyze it more easily. After transcribing the interviews, they were read through multiple times and the answers were formulated as requirements. At this point all requirements were treated with equal prioritization regardless of how common the answer was since it could be the case that a developer just had not thought of that specific solution or considered it as a viable option. Instead, the requirements will be validated further in chapter 4 and 5.

The result of the interviews were the requirements specification which can be found in appendix C and the developers' thoughts on how the prototype best could be implemented. After the interviews, we realized that we received very different answers to our interview questions, which made us realize what a complex problem it really is. When a problem is complex, it follows that there are a large amount of solutions to the problem and depending on where in the CI/CD pipeline developers mainly works, answers can be influenced by what they consider to be important properties for the tool. Furthermore, it may be because different developers use different tools and do not have a good grasp of all the tools that are in the entire CI/CD pipeline. In section 3.3 we will therefore take a closer look at some prominent opinions expressed during the interviews.

Group interviews and brainstorming sessions were also considered as a viable alternative to gather ideas and requirements, but with each developer having an individually and tightly packed schedule it was hard to coordinate, let alone have time to plan for it. However, during lunch breaks and coffee time there was an opportunity to ask questions to a larger group of people, although in a casual non-interview setting. Nevertheless, we often found these discussions useful and enlightening.

Requirements Specification

From literature study, questionnaire and interviews, a requirements specification could be produced as an initial answer to **RQ2**. The requirements specification was developed to form the basis for the prototype that was intended to be designed and implemented in this thesis. In line with how Lauesen [12] suggests that requirements should be categorized, they were assigned different types and levels. We chose to follow Lauesen because he is very descriptive of the whole process of developing a requirements specification. After the requirements had been categorized, they were parted into different groups depending on if a requirement applied to the testing stage, production stage or both in the development process. The requirements were divided into different groups in order to facilitate which requirements to include in different designs.

2.3.3 Design Phase

The purpose of this subsection is to motivate the design phase, why multiple designs where proposed and chosen, what purpose the main design served and lastly motivate why requirements were added during this phase.

Design

The purpose of the designs were twice fold, both to explore designs that could be used when implementing a prototype but also as a way of eliciting new requirements. The design was as such a step in investigating all of the research questions.

There were two main reasons to why several possible designs were proposed. First, by proposing multiple design we were able to suggest several different features and evaluate the developers opinions about them. If we instead had chosen to only make one or two designs we would not be able to evaluate such a breadth of features. Making more than five different designs would require more time, but simultaneously allow us to evaluate more features. Five different designs were therefore made as a trade-off between time spent on designing and the amount of features that could be evaluated.

Secondly, discussing and making the different designs gave rise to new discussions among us about how to best implement the design. The evaluation of the designs also allowed us to get an idea of what would be good or bad to implement in the actual design. These two discussions therefore served as a source of requirements from which further requirements could be elicited.

The designs were discussed with two developers, one of which was our supervisor at the case company and one who is knowledgeable about the topic. The designs and the proposed workflows were then explained to them which gave rise to a lot of discussions about negatives and positives around the prototype.

Main Design

The reason for ultimately formulating a main design was that it could act as a basis for implementing a prototype. The main design was formulated based on the initial requirements specification but also on the new requirements gained from the discussions with the developers. The main design did not have the intention of implementing all of the requirements but

rather a subset of them that we either saw as significantly interesting or were requested by the developers. The main design could have attempted to fulfill all of the requirements specified, but that would require more time, cause the design to be very extensive and difficult and time consuming to implement.

Requirements Gained from Design

Since the requirement specification contained scattered answers of where in the development process developers would prefer the tool to be incorporated and to keep an open mind to how the final tool would be, it was decided to design several workflows for the tool. After discussions with our supervisor at the company and another developer, we came to the conclusions about what was the main design. The main design served as a basis for the prototype and to derive further requirements.

2.3.4 Prototype Phase

The aim of this subsection is to discuss how the prototype phase could be best carried out. The prototype phase had the intention of providing information needed in formulating an answer to all of the research questions. Evaluating the prototype by letting developers test it would help us answer what effect an automated tool would have on the feedback loop and the development process, giving an answer to **RQ1**. When developers are testing the prototype it might be easier for them to identify what information they would like to receive on what worked well, less well and what was missing in the prototype. This would help us answer research question **RQ2**, "What is useful information for the developer, what requirements does the developer have on the tool". By discussing how the prototype would affect the time required for fixing bugs, we will also get and indication to research question **RQ3**. Lastly, it would serve as an indication of how the prototype impacted the sense of closure for the developers, which would help in answering **RQ4**.

Prototype Implementation

The prototype was implemented iteratively, such that an implementation could be evaluated and then improved on in the next iteration. Implementing iteratively would also mean that the implementation would be functional, albeit not fully implemented after each iteration. This allowed us to be agile in our implementation of the prototype. If for some reason we would have to end in the middle of an iteration we would still have the previous iteration to fall back on. Furthermore, reflecting and discussing the previous iteration would also allow us to get feedback throughout the implementation and as such allow us to make corrections to the direction of the implementation.

At last, two iterations were done during the five weeks of prototype implementation. The first iteration was quick, only taking about a week to implement. When it was finished, we reflected on the implementation and discussed what had worked well, less well and what should be done differently.

The second iteration took considerably more time, about 4 weeks. The aim was to acknowledge the advantages and disadvantages of the previous iteration and implement a prototype that would solve these issues.

Prototype Evaluation

In order to make sure the evaluation would work as intended, it was first evaluated with one developer whom gave us input on how to improve the evaluation process. The prototype was then shown and discussed with three different developers in order to evaluate the prototype. As the process of showing the prototype and discussing it with several developers is rather time consuming it will only be done with the final prototype. The prior iterations will be evaluated through our own discussions and reflections as well as by consulting our supervisor at the case company. We could conduct interviews for each iteration, but this would require us to reallocate time and resources from implementing the second prototype. As the time for implementation already was rather limited, it was decided that the first iteration would be evaluated quickly in order to be able to spend more time and resources on the second iteration. There were two reasons to why the prototype was evaluated. First, it would help to validate what requirements were important and if any requirements needed to be reformulated or described more thoroughly. Secondly, evaluating the prototype by letting developers use the prototype makes it easier for the developers to notice what works well, less well and what is missing in terms of requirements.

Additional Requirements

As mentioned in the previous section, allowing developers to use the prototype allows them to notice what works, what does not work and what is missing. The thoughts and opinions of the developers are an indication of what requirements the developers have on the tool. For this reason, these thoughts and opinions are very valuable as a source of requirements for the final requirements specification and should therefore be considered as such. By adding requirements identified during the prototyping phase, we are able to produce a more accurate requirements specification.

Chapter 3
Problem Analysis

This chapter aims to fill our needs of initial requirements that will serve as the basis for the future design as well as to gather a baseline for the research questions. In order to obtain requirements and a baseline, the analysis was carried out in four steps: investigating relevant literature, conducting a questionnaire, holding interviews and a discussion of our own observations. The literature study had the purpose of eliciting requirements and to give us inspiration for possible designs. The purpose of the questionnaire was to both elicit requirements and obtain a baseline for several parameters: feedback, traceability and closure. The interviews aimed at deriving further requirements by interviewing developers which took inspiration from the questionnaire and literature study. These methods were then combined with our own discussions and observations to form an initial requirement specification. This is important in order for the reader to understand how the requirements were elicited and motivate why they were included in the requirement specification. It is also important as the baseline will be used to discuss the results of the research. It could also be interesting to the reader as the requirements specification could act as a starting point would one want to extend or develop the requirements specification further. The chapter will start off with the literature study, followed by the questionnaire analysis, interview analysis, and lastly a discussion of our observations combined with the finding from the previous sections.

3.1 Literature Study

This section aims at exploring relevant literature in order to extract requirements and to get inspiration for the design, based on the experience of others. This is important as it highlighted how similar problems have been approached in other research, which allowed for an indication of where to focus our further investigations. Similar research did also serve as a source for eliciting requirements for the initial requirements specification. The section will first discuss literature relevant to the thesis and then conclude what requirements can be elicited from it. This section will discuss and conclude requirements from relevant fields,

in the following order: performance testing, tests case selection, and feedback loop and bug reporting.

3.1.1 Performance Testing

This section will investigate two articles that researched performance testing, that could give us ideas for how to design a solution as well as to elicit requirements. This is important as it both gave inspiration on how the research questions should be approached, but also acted as a source for requirement elicitation. The articles were found by searching for a combination of the keyword; performance, testing, software, and development, on Google Scholar. First off, we will discuss an article about agile performance requirements specification and testing, and then an article on performance testing and machine learning.

Ho et al. [8] investigated and proposed a framework for specifying performance requirements. Ho et al. mentions that research has shown that performance issues are more difficult and expensive to resolve later in the development process. Furthermore, Ho et al. state that JUnit provides a timeout parameter that can be used to test performance, but also says that complex tests might require probability distribution and that JUnit might be insufficient in those cases. Ho et al. also argues that human testing is undesirable and that successful agile testing relies in test automation. Using JUnit could be a good way of testing performance of simpler methods, especially if the practice of writing JUnit tests is already used during development. But it is also interesting as it would allow for the tests to be configured in code and thereby also be put under configuration management. As mentioned by Ho et al. it is also beneficial to automate the testing, which one could argue is especially important in a DevOps context. From this we concluded that the requirements specification should specify that the tool should be able to be configured in code and be automated. This is specified with requirement **QuDo12** and **QuDo41**.

Hewson et al. [7] investigated performance testing using statistical test oracles. In more detail, they created a framework around JUnit called Buto, which was able to account for differences in the testing environment, such that performance on one machine could be correlated to the performance on another machine. They achieved this by monitoring program resources during run time and used it to generate test oracles. Hewson et al. claims that the model can be used to detect changes in performance and flag future tests that fail. Furthermore, they mention that the way it is implemented causes a very low overhead, but that it could introduce a bias in the test results. This is an interesting possibility that could be used to test performance locally in order to get an indication of how well the system would perform in the production environment. The low overhead could constitute a requirement in a solution, since it would allow for the developer to get a better understanding of how the code would perform in production. This could be a very useful feature. Unfortunately, this requirement was deemed to advanced for the scope of the thesis as well as being outside our area of knowledge. The requirement will therefore not be included in the requirement specification.

3.1.2 Test Case Selection

In this section, an article about test case selection will be investigated.

Test case selection is a test strategy and means that a subgroup of a test suite is selected and executed. Test case selection can reduce the time and money spent testing a system or product. Koivuniemi [10] investigates a test case selection method where test cases are selected based on the files in which code changes have been made. Two areas that were selected to be investigated were feedback time improvement and fault finding capability. The study was performed on two test suites, one with 30 cases and one with 800 cases. The results showed that feedback time improved by 29.3% for the small suite and 55.7% for the large one. For fault finding capability, only the small test suite was examined and the results showed that 97.8% of the failing test cases were captured with a reduced test suite. The article provided inspiration for *how the tool could be designed in the form of tests being selected based on code changes.* The method Koivuniemi suggested could be used in our tool to filter out a developer's changes to remove unnecessary noise from the remaining code base. Since fault finding capability is 97.8% with a reduced test suite, there is a great possibility to still capture all failed tests.

3.1.3 Feedback Loop and Bug Reporting

This section will investigate two articles, one regarding the feedback loop and the other one regarding bug reporting.

Hrusto et al. [9] investigated how feedback between operations and developers could be improved in a DevOps environment. They identified three problem areas that caused problems at a company that had problem with managing and responding to alerts. The first problem was alerts that were not targeting a specific developer, but rather a chat group, which had caused problems with delegating the tasks between the team members. The second problem was described as a low signal to noise ratio, causing important alerts to become lost or normalized and therefore not acted upon. The third and last problem was that a small problem in an external service could cause serious deviations in their service and the feedback from the service provider was lacking. They therefore developed a smart filter, using machine learning, to optimize the alerts that proved to work. What could be of use is the fact that there should be requirements that take these problem into account. The tool should have a way of solving how to target a specific developer as to avoid the first problem. It could also be of interest to have some measure for making sure alerts that are sent from the tool do not disappear in a large amount of other alerts. This resulted in several requirements: **FuPr42** which states that the tool should be able to generate a Jira issue to the specific developer, **QuPr23** which states that the tool should not display entire pages of unmanageable information and lastly **QuPr50** which states that the tool should be able to filter out unnecessary noise from other queries.

Borg et al. [4] researched the adoption of an automated bug assignment, referred to as Trouble Report Routing, at Ericsson. While the case study is not completed as of now, there are valuable insights to be drawn from the report. Borg et al. mentions that there are two types ways of assigning tasks: *push* and *pull*, which could be described as assigning tasks to developers in regards to *pull*, and letting developers choose tasks from a set of tasks in regards to *pull*. They further mention that *push*-based assignment often involves manual work, which they argue is shown to be error-prone, labor-intensive, prone to bug tossing, and potentially results in slow resolution. The concept of trouble routing is relevant to our project as it is important the right developer receives the alerts about performance issues. But also that

push and *pull* assignment have their own benefits and drawbacks that should be accounted for. This resulted in a overarching discussion about requirements and how the feedback should be relayed. If the tool was implemented such that a Jira would be created it should assign the correct developer, which would make the feedback push-based. If the feedback was to be relayed from testing in Github, it would also be considered push-based as the information is relayed to the specific developer. This is related to multiple requirements: **QuPr24**, **FuPr37** and **FuPr42**.

3.2 Setting a Baseline

In this section a questionnaire was used in gathering information with the intention to set baseline measurements and elicit requirements by discussing and analyzing the results from the questionnaire. The initial baselines measurements consist of closure, feedback, issue identification and traceability. This is important because it allows for future measurements to be compared to the initial baseline which can give an indication on the efficiency of the solution proposed in this research. The questionnaire was also important as it allowed us to conclude further requirements based on the current situation at the case company. The questionnaire will be analyzed and discussed, first to set baseline measurements, then to derive requirements.

The baseline from the questionnaire, regarding closure, was that the sense of closure generally increased from the time of starting to implement a feature to the time it has been in production for a week. More interesting was the result that developers in general started feeling, albeit a small, a sense of closure after having had two other developers reviewing the code, which can bee seen in figure 3.1 and table 3.1.

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
After coding	82%	9%	9%	0%	0%
Manual testing, local	27%	37%	27%	9%	0%
Local test suites pass	9%	64%	18%	9%	0%
After PR is created	27%	46%	27%	0%	0%
Built and tested on Jenkins	9%	36%	36%	19%	0%
Reviewed	9%	9%	37%	18%	27%
Merged to main branch	9%	0%	27%	46%	18%
Jira resolved	0%	0%	36%	28%	36%
Deployed to production	0%	0%	18%	55%	27%
Production for a week	0%	0%	9%	9%	82%

Table 3.1: The table shows the respondents' answers regarding closure.

The baselines regarding feedback and issue identification was that developers predominately discover bugs during testing and that the median developer has to fix their code that has been in production, once a month. Although, it is more seldom than once a year that the developer get notified that their code in production is slow. Getting assigned to fix someone else's slow code from production happens every half year for the median developer.

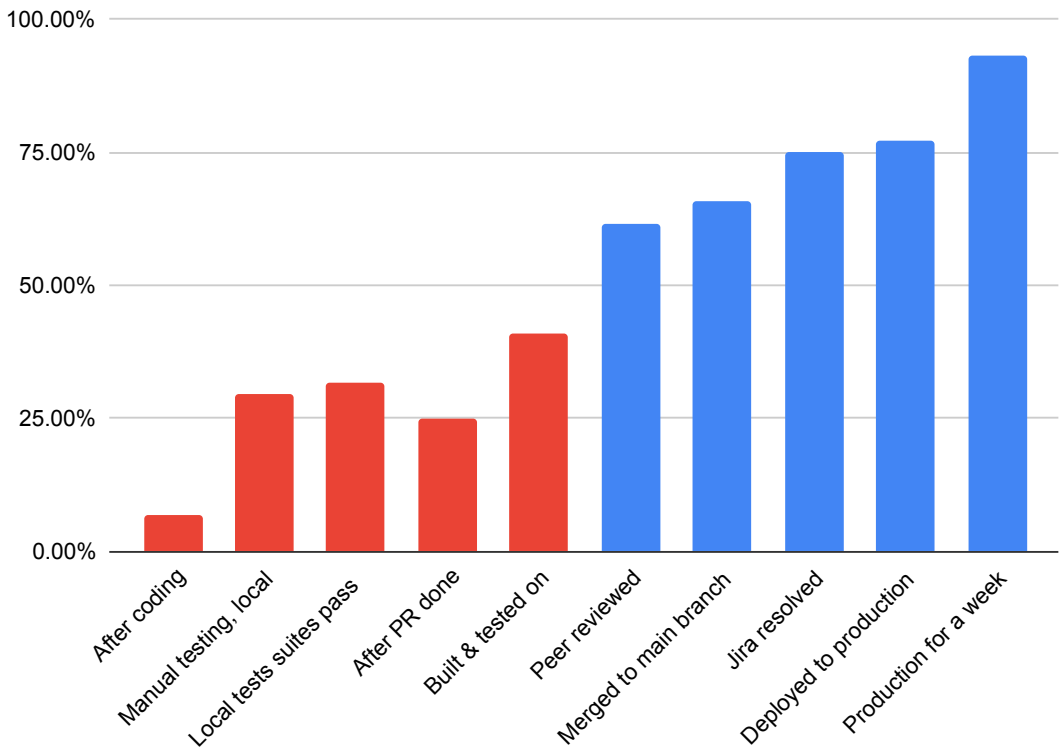

Figure 3.1: The figure shows the developer's sense of closure. The percentage in the table represents the sense of closure, where 0% is *Strongly agree*, 50% is *Neutral* and 100% is *Strongly agree*.

When it comes to the baseline of traceability, the developers opinion about whether the case company has good traceability throughout the development life cycle the most common answer was "agree". The average answer to the statement was, however, closer to neutral. The most common and average answer to the statement of whether it was easy to link an issue in production to a specific code change was "agree". The most common answer to whether it was easy to link a specific code change to a person was "strongly agree", while the average answer was closer to "agree". The responses can be seen in table 3.2.

The conclusions from the questionnaire are thus that developers start feeling a sense of closure after having had their code peer reviewed, that developer sometimes fix their own problems in production but not as often if it is related to performance and that the developers agree that it is easy to trace an issue in production to the person who made the change.

It is logical that developers feel an increased sense of closure the further in the software development life cycle they progress including the longer time it has been in production, since it indicates that the code probably has no severe errors.

3.3 Interviews

This section will focus on analyzing the interviews held with experienced developers and highlight the most prominent areas, with the purpose of eliciting requirements on a possible

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
Telavox has good traceability throughout the software development life cycle	0.0%	20.0%	30.0%	50.0%	0.0%
It is easy to link an issue in production to a specific code change	0.0%	0.0%	11.1%	77.8%	11.1%
It is easy to link a specific code change to a developer	0.0%	0.0%	12.5%	25.0%	62.5%

Table 3.2: The table shows the respondents' answers regarding traceability.

solution. The interviews serve the purpose of investigating the opinion on how to best solve the initiating problem with focus on relating it to the specific context of the case company. The interviews serve the purpose of investigating what requirements a developer has on the design of the tool. Since the developers will be the ones using the tool, it is important that the design of the tool takes their requirements into consideration. The interview questions will be analyzed and discussed per topic in order to elicit requirements. In Appendix B, the interview questions can be found.

At first glance at the interview results, we were able to show that the developers had different opinions about where in the development process they want the tool to detect performance issues. Two developers preferred to get feedback from production as they wanted more reliable feedback on the performance issue even though it took longer to get information. The other two developers preferred to get feedback from the Continuous Integration stage as they wanted to get feedback before their modifications went into production even though that feedback was not as reliable. The fact that the answers are different can be interpreted as the developers having different experience of handling performance issues and therefore based on where the tool best suits their needs. Therefore, a requirement could be that the tool should be able to be *used in different parts of the continuous integration/continuous delivery pipeline.*

There were some differing thoughts about how feedback would be best implemented. Two interviewees thought that feedback would be best received in a pull request on Github, which also aligns well with their opinion that it would be best to receive feedback from testing rather than production, as it would be faster. One of the interviewees believed that faster feedback would be preferred in general but added that it could be hard to get good data from testing and that it would therefore be better to monitor for performance issues in production. The same interviewee also mentioned that code with poor performance that is only executed twice a year are not as important to test or monitor as they do not effect the user experience that much. The opinion of another interviewee was that monitoring for performance issues is good for fine tuning and that such feedback should be relayed in the form of a Jira issue, while testing was seen as a good way of checking the performance quickly. It was also expressed that the tool should be *automated*, since it otherwise probably would be forgotten and never used. Furthermore, one interviewee expressed the opinion the tool should be able to output to the console if run locally.

During the interviews, the interviewees also mentioned several features they would like to have in a tool. One interviewee expressed that there should be a way of *setting a threshold for what should be considered poor performance.* It was not specified whether or not it was per database query or a general threshold, but one could argue that setting custom thresholds manually for queries quite fast could become unmanageable. Another interviewee also mentioned the need to handle both queries that are slow and queries that are not necessarily have poor performance, but are executed often enough to cause issues. One interviewee expressed the opinions that the tool should *be configurable to exclude certain queries, such that they would be ignored.* Similarly, another interviewee mentioned that it should be possible to *specify certain parts of the database that should not allowed to be seized by a slow query.* One interviewee also mentioned that the tool should be able to *distinguish the developers changes and be able to exclude tests that are not affected by the change.*

Several interviewees expressed the need for certain information to be available by the tool. It was mentioned that it should be able to *see the query that was slow, as well as the place in the code it originated from.* Furthermore, there was a need to be able to *receive standard deviation*, min and max time for the queries as well as being able what part of the database is usually afflicted by poor performance. One opinion they all agreed on was the information that one would get from the tool. The tool should *provide information about how long a call to the system had taken, the trace of the call and how long a method call took.*

During the interviews it was also apparent that there were different opinions on how much time the interviewees could imagine spending on testing. One interviewee expressed that a lot of time could be spent on it, while several other interviewees could only imagine spending little time and for it to be easy.

The interviewees opinion on how long they could imagine waiting for feedback differed a bit. One interviewee expressed they could wait about the same amount of time they do today, when waiting on the tests to pass on their build server. Two other interviewees mentioned that they could imagine *waiting 15 and 5 minutes respectively.* One interviewee could imagine waiting one minute extra for the performance tests to run, but also expressed that too frequent tests or reports would eventually become normalized and ignored. It was also mentioned that it is not easy to assign feedback to a specific person, would it be necessary, since it is not apparent who holds the responsibility for a part of the system. One could argue that this is a problem caused by shared code ownership, which is generally considered a positive trait in development. The interviewee, however, expressed that the shared ownership could cause no one to pick up the issue.

Several interviewees said they use the tool referred to as *Query Profiler*, but to different degrees. One used it very seldom, while another used it frequently. This could give an insight into how easy it would be for the development process to include the usage of a manual process to measure performance.

In regards to writing tests, some developers seem to write tests often, while other seem to write tests less frequently. If developers generally had adopted TDD, it could be beneficial to *include performance testing that is similar to writing the current tests*, as it would not be a very large change in the development process. This corresponds well to the fact that three out of four interviewees expressed that they would rather *test performance by writing tests* rather than measuring it using a GUI. One developer mentioned that neither was preferred and that the tool should instead be *automated.*

3.4 Requirements Specification

This section will analyze and discuss results from previous sections together with our own observations to elicit requirements. Similar requirements were identified in several previous sections and should therefore be discussed in terms of how they relate to one another. The discussions were then analyzed in order to derive further requirements. This is useful as it will serve as the starting point for the upcoming design in the report, but could also act as a starting point or inspiration for making your own requirements specification. This section will start off by analyzing the requirements found in the previous sections and then combine them to form a requirement specification. The section will group similar requirements together based on the literature study and discuss them one after another. All requirements can be found in the Appendix C.

In the interviews, 3 out of 4 developers preferred to test performance issues in code instead of GUI. Then, as Ho et al. [8] writes, JUnit tests with time-out could be a requirement for the tool. With a time-out, developers can set the desired length for how long a method call may take. However, JUnit seems to be a bit limiting because developers also wanted the tool to show the entire stack trace of a method call, something that JUnit does not. Regardless, *JUint* could be a requirement and must therefore be included in the requirements specification.

In Hewson et al. [7], they present a framework that senses and adapts the performance testing to the hardware. Although, this approach could be a possible requirement for the tool, eventually we considered that it was out of context for this research.

From analyzing the interviews, it could be concluded that two developers preferred to test only their own changes or relevant parts of the system for performance issues. The just mentioned requirements that the developers place on the tool go well with the method that Koivuniemi [10] uses in his research. Koivuniemi also claims that the fault finding capability is very high even if a smaller test suite has been used and then one could expect the method to find the performance issues that a specific code change provokes. A developer also set the requirement that it should be possible to both exclude and include certain tests, which could be possible with test case selection. Therefore, we chose to add *test case selection* as a requirement.

In the interviews, one developer said that when there are many developers working in the same layer in the stack, it is difficult to know who is responsible for a performance issue. Then you instead send a performance issue to the responsible team and the members can investigate who is responsible. Hrusto et al. [9] claims that it can be a problem when delegating responsibility within a team and the issue can fall between the cracks. It could have the effect that performance issues remain in production longer than necessary and affect the customer. Therefore, we believe that a requirement for *automated traceability should be that the tool provides feedback to the developer responsible for the performance issue.*

Furthermore, the same developer said that if you get constant feedback on performance issues that are not relevant to the developer, it can be perceived as noice and over time, developers may ignore the feedback. This was also known to Hrusto et al. establish where they also considered that frequent problems were normalized and thus took no action. From this analysis, requirements where the tool would *not provide repetitive information about the same performance issues could be deduced.*

Developers also emphasized in interviews that they wanted feedback in pull requests re-

garding performance issues. Getting feedback on performance issues in a pull request makes it easier for the right developer to get information about issues through push assignments. As Borg et al. [4] mentions, it is otherwise easy for issues to be sent around and a solution to issues is delayed. Two other developers mentioned that they preferred to get feedback in the form of a Jira issue where the responsible developer is assigned automatically. Through this analysis, it contributes to the requirement that *feedback should go directly to the responsible developer*.

Some requirements in the requirements specification were not a results of the previous literature study, questionnaire or interview alone but rather our own discussions and ideas. New requirements were formulated and added to the initial requirements specification. It can be found in Appendix C.

Chapter 4

Design

This chapter will focus on investigating possible design solutions, discussing the designs, elicit additional requirements and lastly motivate the design choice. As different possible solutions are explored, this will help the reader understand benefits and disadvantages of the design solutions, how the initial requirements specification from the previous chapter was improved, and lastly how the design choice was made. The design choice is especially significant, as the design will serve as a basis for the prototype in chapter 5. This chapter could also be interesting to the reader as a starting point if one would want to implement a similar solution. First, the design proposals will be explored together with a discussion of advantages and disadvantages of each design solution. In the second section, additional requirements will be derived and discussed. Lastly, we will motivate the design choice based on the initial requirements from chapter 3 and the additional requirements identified in this chapter.

4.1 Design Proposals and Discussions

This section will explore different design possibilities based on the initial requirement specification, elicited in chapter 3. With each design proposal follows a discussion about advantages and disadvantages of the design together with opinions from developers as well as our own. From the discussions additional requirements will be derived. This will give the reader an overview of the considered solutions and their strengths and weaknesses, which will be important in order to understand the rest of this chapter. The designs will be explored one by one where we will discuss the advantages and disadvantages of each design proposal as well as opinions from developers and us and finally develop new requirements for each design proposal.

4.1.1 Design Option 1

The first design option, as seen in figure 4.1, is based around monitoring the production environment and automatically relay Jira issues to the original developer. The developer thus pushes new code to Github which then will be built and tested in Jenkins and then released into production by a developer. The solution will then monitor the production environment in order to find slow queries. The solution will measure and find were in the stack trace the software spends the most time. From the stack trace the location in the code would be derived. In order to find the person responsible for the code, the solution would look who last changes it and create a Jira issue assigned with the developer. This design is based on section C.1 and C.3 in Appendix C.

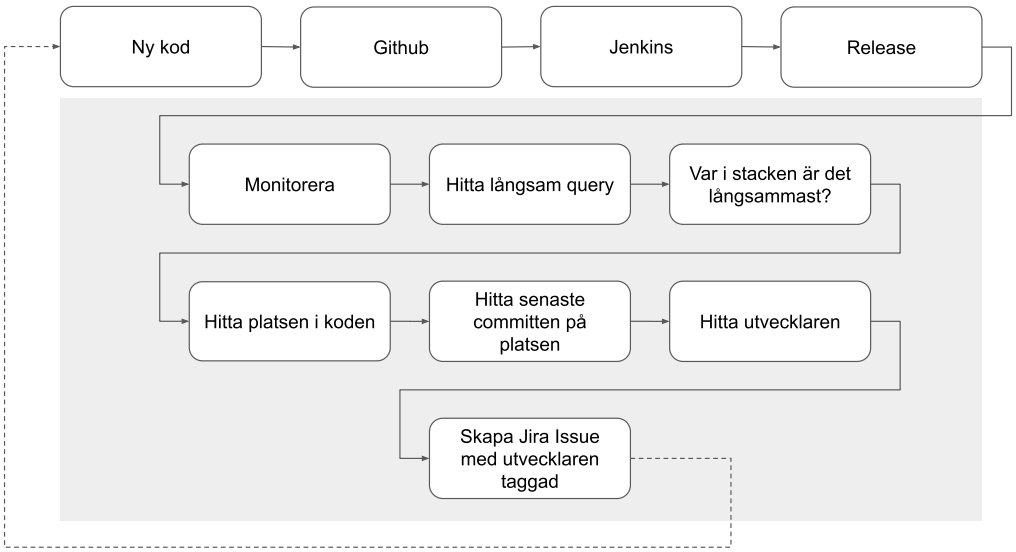

Figure 4.1: Design option 1

This design has several benefits, but also several drawbacks that might outweigh the benefits. As the design is based on monitoring the production environment it will result in more accurate results. Compared to performance testing where you have to make sure the environment is as similar to the production environment as possible in order to get accurate results, the solution relies on the production environment itself. This also makes it easier to implement as you do not need an additional server identical to the production server, which might be very costly. Also, implementing the tool should be relatively simple as the steps required are not too complex to implement. The simplicity of the design and functionality might, however, be one of its weaknesses.

There are several issues with the simplicity of this design. First of all, since the changes have to be deployed into production, the feedback takes several hours or days to receive. Also, assuming that the longest call in the stack trace is the cause of the performance issue might not be true as it could also be caused by a substantial amount of fast calls. It is also not necessarily true that the latest change actually caused the performance issue. For example, if the code has been changed multiple times, it is not easy to derive what change actually caused the issue,

let alone automatically. Creating a Jira issue every time a performance issue is detected could also cause trouble as it would create duplicates and would flood Jira with unnecessary issues. It should therefore have some measurements on whether the same issue has been added into Jira before. Also, assigning the Jira to the right developer is a very complex problem that is not trivial to implement.

The requirement specification should therefore include a requirement that the tool should *only create one Jira issue per performance issue.* It was also evident from our discussions that the tool should be able to *integrate with Jira, such that Jira issues can be created automatically.* The initial requirements specification had not specifically specified that the tool should be able to *monitor the production environment*, which is an important requirement for solutions requiring the production environment to be monitored. From the discussions with developers, it was also brought into light that finding the right developer for the Jira issue is not easy and that this design therefore can be difficult to implement.

4.1.2 Design Option 2

This design is similar to the previous design, with an added layer and can be seen in figure 4.2. The code would be pushed to Github by the developer, built and tested in Jenkins and then a release would be built. The build would then be deployed from the master branch as a beta version of the software, which is only accessible to customers in the beta testing group. The time of the deployment will be recorded in a database, along with version of the beta release. The developers will be notified in the chat that the beta has been released successfully. As with the previous design option, the beta version would be monitored to find slow queries by monitoring how much time the software spends in each part of the stack trace. The location in the code can be derived from the longest stack trace and the developer that last changed the code can be found using Git. If one or several slow queries have been detected during the beta testing, Jira issues will be created and assigned to the correct developer. If no slow queries are found, the beta version will be released as an actual release. This design was not strictly based on the initial requirements specification, but rather as a discussion of how to solve the initiating problem.

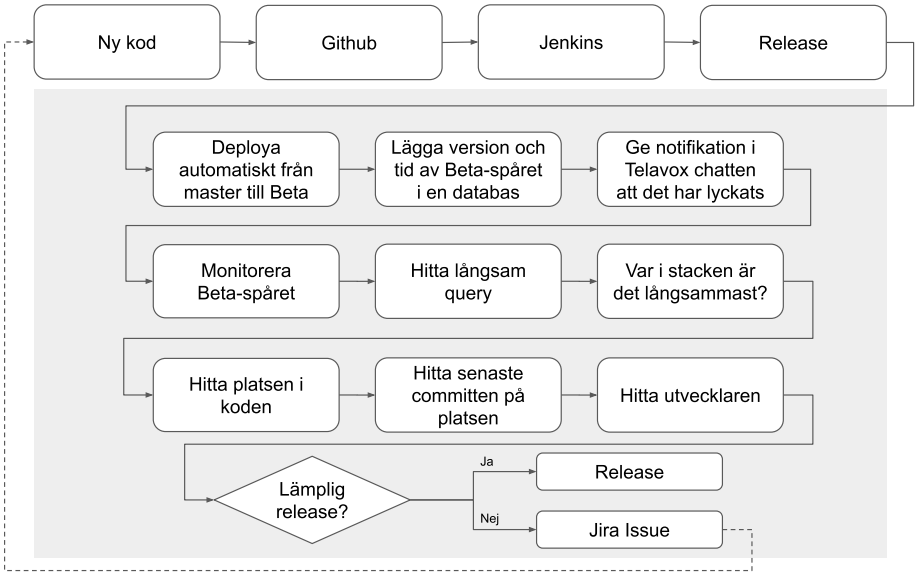

Figure 4.2: Design option 2

This design is similar to the previous design, with the difference being that it is extended to first test the code in a beta testing stage. It therefore has many of the same benefits and drawbacks discussed in the previous design.

As mentioned earlier in this section, this design was not strictly based on the initial requirements specification, since it did not mention a beta testing stage. Rather it was discovered as a possible solution by us discussing the initiating problem, that the user experience could be improved by making sure less performance issues reach the users. There are several reasons to why beta testing is not supported by any requirements gathered earlier. It could be that some requirements were overlooked in the problem analysis phase, but it could also be the case that it is not actually a requirement. The initiating problem was to investigate how to provide better feedback to developers on finding and solving performance issues and thereby improve user experience. A goal requirement that would both support the beta testing phase and support the initiating problem could therefore be that the tool should *reduce the amount of performance issues being released into production and the time they are in production.* We found it interesting to know whether or not it would be beneficial to have a beta testing phase and it was therefore added as an extension of the previous design.

However, the added advantage of testing the code in a beta stage before releasing the code into production is that the end users are more protected from performance issues.

The added benefit, however, comes with its own drawbacks. First of all, since the code first has to pass through a beta stage, the feedback from production is much slower than previously. The feedback from the beta stage could, however, replace the need for feedback from production. But, the size of the beta stage might impact the amount of feedback developers are able to receive from it. Adding a beta stage also makes the design more complex to implement into the development process as well as delaying any changes or features added since they first have to pass through the beta testing phase.

From our discussions it could be beneficial to add requirements that specify that the tool should incorporate beta testing, as well as it being able to send notifications to the internal

chat service at the case company. When discussing the design with developers, however, it was revealed by the developers that incorporating a beta stage would be very complicated and not add much to the solution. It was also brought to light that the tool should not send messages through the internal chat service at the case company, since the developers believed it would be enough to receive feedback in either Jira or Github and that also sending the same feedback in the internal chat service would be redundant or even confusing. The developers also acknowledged that it should be possible to find who changed a part of the code most recently, but that would not necessarily be the right developer to assign an issue to.

4.1.3 Design Option 3

In this design, as can be seen in figure 4.3, unit tests with time-out was introduced in the Continuous Integration stage. Unit tests with timeouts are regular unit tests with a time out value, which will cause the test to fail if exceeded. Developers would implement new tests and code to then run tests on their local machine to make sure that the tests pass. If any tests fail or takes longer than the threshold time, the time out, developers get an indication about where in the code there is a performance issue and go back to implementing new changes. If tests pass, developers can commit and push their changes to Github. In Github, developers make a pull request to the main branch. A pull request triggers Jenkins to build and test the code changes. If the tests fail, developers get information about what test that fail and in that way an indication of about where in the code base the issue can be. If the tests succeed, code changes can be merged into the main branch. From the main branch, developers can build and deploy the new changes. The design is based on requirements from section C.1 and C.2 in Appendix C.

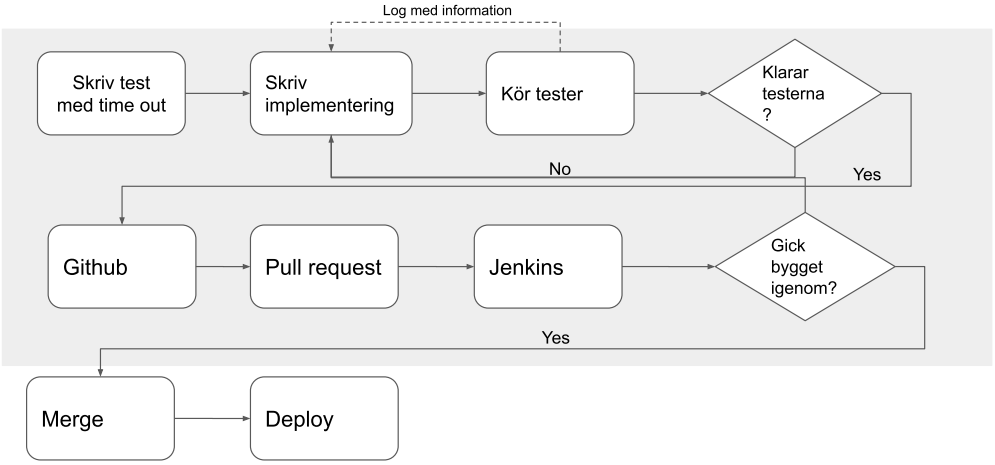

Figure 4.3: Design option 3

This design is different from the two previous designs as it relies on testing the system rather than monitoring the production. While the case company already relies on testing to

make sure their system works as intended, they do not measure the performance of it prior to deploying it to production. The benefits with this design is that it uses the unit testing framework that the case company already has implemented to implement performance testing by setting a time out to the already existing unit tests. As the software can be tested for performance issues on the local machine or on the Jenkins server, the feedback is faster compared to monitoring production. An added benefit of it being able to be run on a Jenkins server is that it is relatively easy to implement this testing as a check in a Github pull request. There is also an advantage in terms of speed. Unit tests are generally short running and as this design is based on writing and running unit tests with time outs, this sort of testing should be of similar speed to regular unit testing.

The disadvantage with this approach is that the option relies on testing rather than monitoring, which has the implication that the testing environment has to be very similar to the actual production environment in order to be able to produce accurate results. Since neither the local machine or the testing environment on Jenkins is setup to be similar to the actual production environment it could be argued that the reliability of the results is low. However, even though the reliability might be low, it might give the developer an indication of whether or not the code performs well. Since unit testing is not specifically designed for testing performance, it can be insufficient for testing more complex features. The information from the tool is also limited as it only specifies whether or not the test passed or not based on a threshold time, which might make the culprit hard to discover. It could also be the case that failing tests become normalized due to various reasons: tests that fail regularly will be ignored, test that are not relevant to the developer at the moment could be ignored and a lot of failing tests might be difficult to act on for a developer. Running all performance tests all the time might also be too slow to provide fast feedback to the developer, but it depends on the amount of tests and the tests themselves.

From this design option we could identify a few requirements that would be beneficial for the final design and implementation of a prototype. Tests should be able to be made by *adding a time out to a unit test.* The tool should be able to *give information of what caused the test to take too long time.* The tests should be able to *be written using a widely used testing framework*, such as JUnit or TestNG.

From interviews, however, it was mentioned that the developers have to learn how to write a new type of tests and that getting developers to adopt the practice of writing performance tests is difficult. It was also mentioned that the feedback from the tool should contain information of how long each call in the stack trace took to execute, as it would make it easier to debug.

Concluding our own and developer's thoughts on the design, it was apparent some additional requirements were needed. The performance tests should be *written using a standard testing framework such as JUnit or TestNG, combined with their time out functionality.* The information should *include how long each call in the stack trace took to execute.*

4.1.4 Design Option 4

This design option is based around unit testing for testing performance combined with test case selection and can be seen in figure 4.4. The developer would start by implementing a feature or function in code and then write tests for it. The tests would then be assigned to a relevant test category, which can be used in order to specify what tests should be run.

The developer would then run the test category locally and thereby be informed of what test pass and fail. If the tests pass, the developer would create a pull request that would trigger Jenkins to pull the new changes and test the performance on the Jenkins server. If the tests pass, the new changes will be merged into master and then released into production. In this option, the design has been based on C.1 and C.2 in Appendix C. More specifically, this is a way of filtering unnecessary noise from the feedback, requirement **QuPr50**. But also that the developer should be able to filter out performance tests that are unnecessary to test, requirement **FuPr45**.

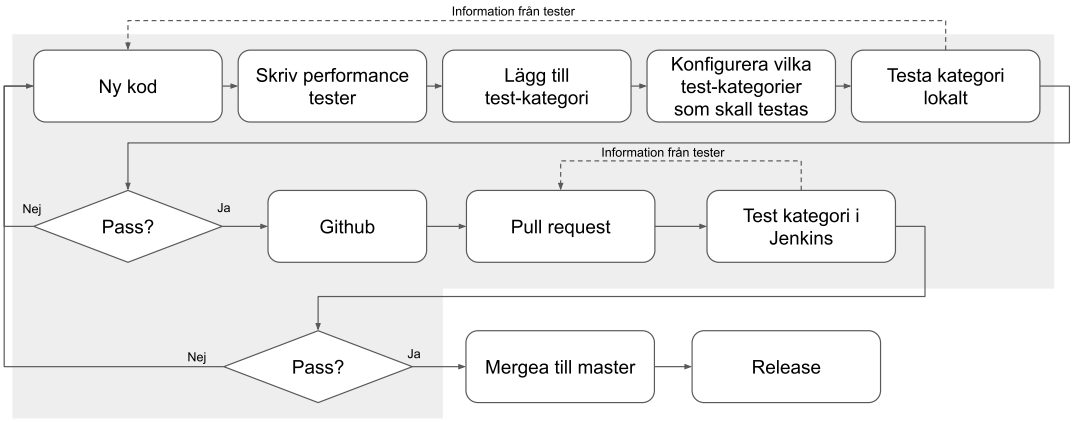

Figure 4.4: Design option 4

This design is similar to the design in 4.1.3 but instead of a time out for each unit test, performance tests are implemented and test case selection is introduced for performance testing in the form of dividing all tests into categories. By categorizing performance tests, developers can run the categorizes relevant to their changes and thereby only testing for performance issues for their own changes. This design offers other advantages and disadvantages compared to the design in 4.1.3 which leads to new requirements being derived. An advantage is that developers can put their performance tests in a test category based on what they want to test and then only run performance tests in that category. Running fewer performance tests take less time and as such it gives faster feedback to the developer about possible performance issues. Another advantage of this design is that categories can be run both locally and on the Jenkins server, since the tool is compatible and able to be incorporated in several stages of the development process. This means that the code can be checked several times in different environments and thus ensure that performance is satisfactory in different conditions.

In addition to advantages, several disadvantages could be identified for this design. The first disadvantage was that test case selection may not capture all failed tests because the method only runs subsets of all performance tests. This can lead to developers introducing a performance issue in the code base that they are not aware of and the changes are then pushed to the common repository. Then it can affect other developers and it becomes difficult to

track it back to the responsible developer. Another disadvantage is when performance is tested in the Continuous Integration stage, there is a risk that it does not correspond to how the code would actually behave in production. This can result in performance issues not being detected before they end up in production. However, it is always an advantage to test in continuous integration to get an indication of how the performance of the system is. Another disadvantage may be that the tests that a developer wants to run are not in the same test category and thus several categories will be run. This could mean that, eventually, the entire test suite would be run and test case selection has no function. A final disadvantage could be that the different test categories become difficult to manage, as it can be difficult for developers to keep track of what categories there are and what tests they include. A result of unmanageable tests categories is that it can make it difficult for developers to know which categories they should run to cover the code changes they have made.

The additional requirements we could establish from this design were that the tool could *run single or multiple categories, be able to add categories and add tests to the categories, a test should be able to be added in several categories and the workflow should be fast enough for Test Driven Development.*

After showing this design to developers, they expressed their opinions about it. One opinion was that they wanted stack trace with time to be included in the design as they thought it was important to know what in stack trace that took a long time. They also mentioned that categories would be difficult to keep track of as they would often have to go into the different categories to see what tests were available and it would provide a lot of extra work. Finally, they mentioned that categories will contain tests for code that a developer has not changed, which would give irrelevant feedback and be perceived as noise.

From analyzing the interviews, it could be concluded that an additional requirement would be that the tool should *provide a visualization of the timed stack trace.*

4.1.5 Design Option 5

The general idea of this design was to configure what tests should be run with the use of a configuration file, but more significantly the design would measure the stack trace for each of the tests using a script file. The design option is shown in figure 4.5 The developer would therefore first write a unit test, that fails, and then write the code. The configuration file would then be edited such that the relevant tests would be tested and measured. If a tests takes longer than a specified threshold time, the test will fail and the time the test spent in each part of the stack trace will be printed. The code and the configuration file are then pushed to Github where a pull request is made. The pull request will trigger Jenkins to pull the code and run the same test on the Jenkins server. If all the tests pass, the code is merged into the main branch and then deployed. This design option is based on requirements from section C.1 and C.2 in Appendix C.

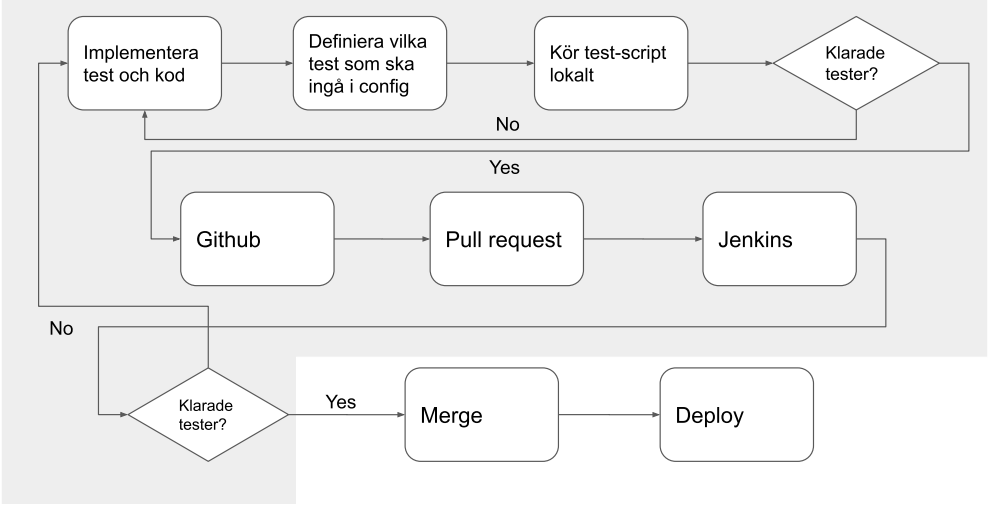

Figure 4.5: Design option 5

This design option is similar to 4.1.4, with the difference that the test selection is configured using a configuration file and then executed using a test script which will also measure the time the software spends in each part of the call stack. As the other designs 4.1.3 and 4.1.4 do not offer the whole call stack so our idea was that through the script, each performance test case could be executed while the call stack with time measurements was saved. The advantage of this design is that the developers themselves can specify what tests should be run. This gives them more control over the testing and can also make the testing faster as a result of less tests being executed. As with the two previous design options, the testing can be carried out both locally and on a Jenkins server. Since the design would measure the time the software spends in each part of the call stack, the developer can more easily see what parts of the system is causing it to be slow.

The downside with being able to specify exactly what tests you want to run is that you have no choice but to decide what tests to run. This creates an extra manual step that complicates the workflow and introduces room for mistakes. As there did not exist any form of feedback about performance in Continuous Integration stage, it was considered that the time that the extra manual step was worth it to receive quicker feedback. Measuring the time for each part of the function call might introduce its own bias and skew the results, making them less accurate.

The tool should therefore be able to, *automatically, identify what tests to run based on the changes the developer has made.* However, the developer has to be able to *configure specifically what tests to run if needed.*

During the discussions with developers it was also mentioned that it would be beneficial in terms of bug finding to be able to *see the stack trace with each call measured.* Using a file to configure what tests are executed was seen as a odd way to configure the tests since the file would be changing constantly. The tool should therefore be able to *measure each call in the stack trace.*

4.2 Requirements Gained from Design

This section will reflect on the work carried out during the design phase and discuss the requirements elicited from the previous section. By discussing the designs with the developers and between us we were able to narrow down the requirements specification, based on what was needed at the case company and what would be interesting to experiment with in the prototype. In this stage, it was also possible to gather requirements that specified the backend or hardware part of the tool. First, this section will reflect on the useful requirements gained from the previous section and then discuss the reason they were added.

One area that became clear during the interviews that was important for the developers was that the tool should be automated and a number of new requirements emerged. One requirement that emerged was that the tool should be *integrable with Jira so that it could automatically generate Jira issues when a performance issue was detected.* This requirement mainly apply if the tool is based on monitoring the production environment. Another requirement that emerged was that the tool would *automatically identify which tests would be run based on the code changes that had been made* and this requirement mainly applied to the different test environments that the developers have access to. We chose to add these requirements as the developers mentioned it repeatedly during the interviews when we presented the different designs. It should be noted that this requirement is similar to **FuPr43**, but different in the sense that it does not specify that it should be an automated process. The reason for not identifying this need earlier could be due to the reason that it was not obvious that this process should be automated. After discussions between us, we considered that if these areas were not automated, it would create extra work for the developers that can feel tedious and be time consuming. When it comes to feedback, the developers wanted a *Jira issue per performance issue to be created.* After discussions, we considered it obvious to add it as a requirement. If developers are constantly flooded with new Jira issues, it can be perceived as noise, which leads to developers ignoring them and the tool thus loses its purpose. It would also fill the backlog with duplicates of tickets that would create extra work by having them removed and other tickets disappearing into a sea of automatically generated issues.

The developers emphasized that one of the most important parts of the tool was to increase the traceability in performance issues by showing the stack trace together with the time for each method call for each performance issue. The developers can then see where in the stack trace it is slow and examine it more closely if it concerns their code changes. If it is not their code changes, they can put a ticket in Jira. Since the developers placed great emphasis on *increasing the traceability of the stack trace*, we chose to add it as a requirement. It also emerged from our own discussions that if you want to know where in the code a performance issue arises, it is important to be able to *measure the time for the various method calls.*

Further requirements that emerged from discussions with the developers were that *tests should be written with a standard test framework* so that developers do not have to familiarize themselves with a new framework. The actual test phase should be fast enough to be used in Test Driven Development (TDD) and developers should be able to *specify which tests to run if needed.* Adding a new test framework they thought would make the test phase unnecessarily complicated and could also affect TDD by taking more time than necessary. By being able to specify which tests are run, the feedback time for TDD could be shortened. As these requirements advocate that the tool be kept as simple as possible, we considered that they should be added to the requirements specification.

The new requirements were added to the requirements specification and the updated version can be found in Appendix D.

4.3 Design Choice

This section will motivate the choice of design and aims at scoping what parts of the requirements should be investigated further in the prototyping chapter 5. Since the design will form the foundation on which the prototype will be implemented, it dictates what requirements the prototype will implement. The reason for scoping the requirement specification is that it currently does not specify a single tool, but rather specifies a range of requirements that caters to different tools and methods of solving the initiating problem. As such, the requirements specification will go from a set of general requirements regarding the issue, to an unified specification for the specific design. We will first explore the chosen design, shortly motivate how it relates to the design options and then motivate the choice.

The chosen design will be based on performance testing, but with the difference that it will rely on already existing unit testing rather than relying on developers writing new tests. The unit testing will run while being monitored, such that the call stack can be measured and given to the developer as feedback. The design should be able to run both locally and on a Jenkins server. This design is similar to design option 5 in 4.1.5, in that the tool will automatically measure the test suite. The difference is that design option 5 relies on the developer writing new tests specifically for testing performance that will be measured. The chosen design use the already existing unit tests instead.

The reason for choosing this design is that it does not require any extra work to be done other than to set up the monitoring such that the existing tests are monitored. Since the unit tests are not specifically designed for testing performance, the coverage and reliability of the results can be argued. However, it might be able to indicate whether or not there is a performance issue in the code. During the discussions with the developers it was mentioned that a solution that would not require much additional work is preferred. Since the design does not require the developer to write new tests, it is very cheap and easy to implement given that unit testing already exists. The trade-off between cost and benefit was therefore one driving factor in making the design decision. The reasons for choosing a design that relied on testing rather than monitoring the system in production was that the feedback loop becomes much shorter, possibly even making it compliant with TDD practices, but also as it makes the process of routing an issue to a developer much easier. Using this approach, the developers would make sure the performance is good enough themselves which alleviates the need for routing issues to developers. Another reason for monitoring tests rather than writing new tests is also that the monitoring would allow information like call stack execution time to be gathered and presented to the developer, which could aid in resolving performance issues. It should, however, be added that this design does not in any way hinder from also monitoring the production environment. These two methods of identifying performance issues are in many ways compatible with one another. However, in order to limit the scope of the remaining thesis, we will focus on testing performance solely. The reason for implementing a prototype is two folded. The implementation itself will validate the requirements in the context of the chosen design and elicit new requirements based on our experiences of implementing it. Also, by allowing developers to use the prototype, they may be able to recognize require-

ments that they would not otherwise be able to recognize from only discussing the designs. Furthermore, using the prototype shows the implementation of the requirements which can help validate them, since their presence or absence might be more noticeable. The chosen design will be implemented in the following chapter, in order to validate the requirements as well as to elicit new requirements.

Chapter 5
Prototype

This chapter will validate a part of the requirements specification and elicit additional requirements by implementing and evaluating a prototype. By evaluating the prototype we will also gain insight on the developers thoughts and opinions regarding an automated tool. This is important as it will validate that the requirements elicited throughout the project reflects the actual needs of the developers. Furthermore, eliciting new requirements from the implementation and evaluation allows the requirements specification to be improved in terms of completeness. The reason is that trying a prototype out facilitates the developers' ability to notice what works well, less well and what is missing. The chapter will start off with exploring the implementation in two iterations and the experiences gained from implementing the prototype. The prototype will then be evaluated by interviewing developers, followed by a discussion about the requirements gained from the prototype.

5.1 Prototype Implementation 1

This section will discuss how the first prototype was implemented, motivate the implementation choices and discuss requirements identified during this iteration of the prototype. This is important as it motivates the decision to make the second prototype as well as the direction of the second prototype. First, an overview of how the prototype was implemented will be explored, the implementation choices will then be discussed on a more granular level and then lastly, the requirements identified during the prototype implementation will be discussed.

The purpose of this prototype was to implement a solution for measuring how long the java code spent executing different method calls. The idea was to modify every method call in a java program, such that the start time and stack trace could be logged before the actual method was called and then once more log the time once the method was finished. This would theoretically allow every method call in a java program to be logged, in order to produce a timed stack trace. Java, fortunately enough, has an API that allows a java agent to be attached when running a java program. A java agent is able to define a method that will be run prior to

the java program starting, called the *premain* function. But, more interesting is that the agent is able to *transform* all the methods during run time, with the function *transform*. Using a java library called ByteBuddy we developed a java agent that could *intercept* all the methods calls and alter them during run time, with the method called *intercept*. The method calls where changed such that a timer was started at the beginning and then stopped at the end of the method in order to time the method. The execution time and stack trace was dispatched to another thread, in order to not influence the execution time of the original thread too much. The result was that each time a java program was run with the agent, a json log file was produced which showed how long each method call took and the stack trace at that point. The prototype was then further developed such that it would load the current git repository and only *intercept* the methods that were in one of those files. The result was an agent that could be attached to any java program, that would produce a log for each java class file that was run and had changed. This could then be run in combination with the unit testing such that tests would be measured during runtime. The generated log could then be measured in order to find whether any performance issues had occurred.

There were however some downsides with the prototype. There were no way of visualizing the log and it was hard to read as it was. We would either have to use an already existing framework for visualizing data or make it ourselves, if we were to pursue this idea further. Furthermore, the behavior of the agent was unpredictable when running the more complex programs or multi threaded code.

When we were done implementing this prototype we argued that we could not have been the first in trying to accomplish this. Our thoughts were validated rather quickly. Many of the monitoring software used for java use the instrumentation API by implementing a java agent that sends data to a database. This is for example the case with the monitoring software Elasticsearch and Kibana. The idea of using a java agent in order to extract information during run time made us realize that off the shelf monitoring should be able to monitor tests, also.

5.2 Prototype Implementation 2

This section will discuss how the second iteration of prototype was implemented, motivate implementation decisions and discuss the requirements gained during this iteration. This is important as it will help the reader get an understanding of both how the implementation was implemented and furthermore the reasoning behind the choices made throughout the implementation. Similar to the previous section, the overview will first be explored followed by a discussion of the prototype on a more granular level. At last, requirements identified from this iteration will be discussed.

The first step in implementing this prototype was to find a suitable monitoring tool. It was mentioned in interviews that the case company used the ELK stack tools from Elastic for monitoring in production. Since developers already had some experience with the ELK stack tools, we considered that those would be suitable for the prototype as developers might get an easier grip on the prototype. The tools from the ELK stack that was chosen for the prototype were Apm server, Elasticsearch and Kibana. In addition, a Java agent was hooked on to the Java system.

In order to provide the same set up of the Apm server, Elasticsearch and Kibana to all

developers, it was decided to use Docker. The Apm server, Elasticsearch and Kibana run in separate Docker containers and together they formed a Docker network where the containers can communicate with each other. By using Docker, we could save some time that would go to installing and setting up the different tools locally. Furthermore, if changes to the set up needs to be done, new images for Docker can be created and deployed to developers and everyone will easily be updated with the latest changes. We also chose to add Jenkins as a container to the Docker network to save time for setting one up and to clarify to developers what the course of event looks like.

To facilitate the implementation of the prototype, we decided to use a smaller application than the one at the case company. Since a larger application makes it more difficult to troubleshoot what has caused errors as it contains more components, we chose to implement the prototype around the smaller application. The smaller application was based on the same programming language and build tool as the case company's where it was programmed in Java and used Maven to build the application. One difference was that it used a different test framework, but we considered that there was no decisive difference for the prototype as the Java agent was not dependent on a particular framework. To be able to monitor the tests, the agent was hooked on the Java application by being added to the repository and to the Maven build file. When the application was built the agent is included and records what methods in the application are being called during run-time and testing and gather data about them. This data is then sent to the Apm server.

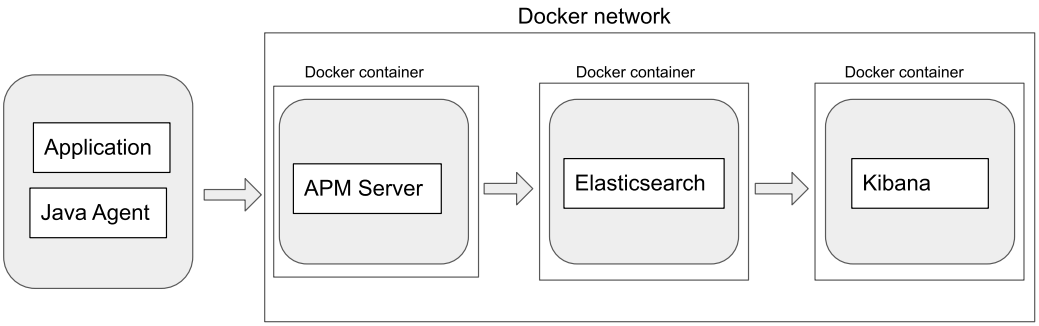

Figure 5.1: Architecture of prototype for local testing

The prototype was implemented for two scenarios. The first scenario, which can be seen in figure 5.1, is for when a developer wants to test for performance issues on their own machine. First, they start by launching the Docker containers and network. Once that is done, they can implement tests for the changes in the code base that they want to make. Then they implement the changes in the code base and can run the test locally. When the test is run, the agent captures the method calls along with how long time each call takes and forwards it

to the Apm server. Further, the Apm server forwards the data to Elasticsearch. Elasticsearch works as a database where all data is saved and it is possible to connect several Apm servers to it. Kibana then retrieves data from Elasticsearch to visualize data to users. After all tests have been run through, the developer can go to Kibana to get feedback on whether there are any performance issues in his/her changes. If performance issues are detected, the developer can display the call stack and trace where the performance issue is located and then improve the code to exclude the performance issue.

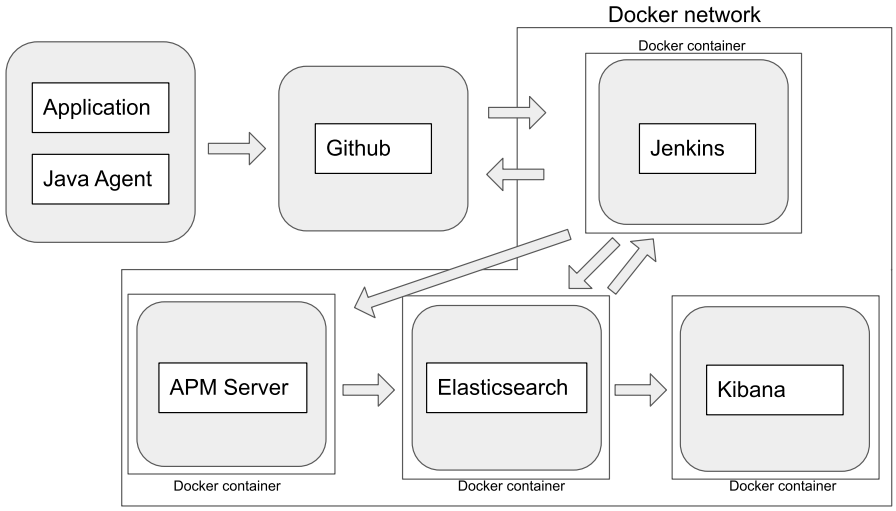

Figure 5.2: Architecture of prototype when running on build server

In the second scenario, which can be seen in figure 5.2, developers can test for performance issues during more extensive testing on the build server. The developer pushes up his/hers code changes and tests together with the Java agent to Github on a separate branch in the shared repository. From there, the developer creates a pull request to the main branch. By creating a pull request, a webhook is triggered for Jenkins to download the new changes, build the application and test it. When tests are run on Jenkins, the Java agent will capture the method calls and send data to the Apm server, which in turn forwards it to Elasticsearch. When all tests have been run, Jenkins makes a query to Elasticsearch to investigate if there are any performance issues. If performance issues exist, Jenkins gives a red cross in Github and Github provides feedback to the developer that there are performance issues in the code. The developer can access information about performance issues by visiting Kibana and can search for a specific commit id to get information for a certain code change. When the code change has been found, the developer can trace where in the call stack performance issues have arisen. Based on the information, the developer can optimize the code to be able to exclude performance issues from the code base.

During the development of the prototype, we encountered a few difficulties. Most of the difficulties depended on the lack of experience from our side, hence a large amount of time was spent learning how things worked. One example is setting up Docker as it was a bit complicated to set up the Docker network and get the different containers to communicate with

each other. It also affected the communication between agent, Apm server, ElasticSearch and Kibana because they were dependent on Docker. After these parts worked, the next struggle was to get the java agent to send data to the Apm server. From having debugged a lot, we could conclude that the application was started on a different thread than the agent monitored and thus could not intercept the data. Another problem we encountered was when retrieving data from ElasticSearch. Our experience in combination with a deficient manual meant that we spent a lot of time trying to retrieve data from ElasticSearch. In addition to the problems mentioned above, the implementation went well. For example, Jenkins went smoother than we thought to set up, thanks in large part to detailed guides available online.

5.3 Prototype Evaluation

This section will investigate and discuss the results from the evaluation of the prototype, which were gained by evaluating the prototype with developers. Discussing the results from the evaluation will help the reader understand the conclusions drawn from the evaluation of the prototype. It will also validate the requirements that were incorporated in the design and prototype as well as to elicit additional requirements. First off, reoccurring topics from evaluation will be discussed and then followed by conclusions from the discussions and identified requirements.

5.3.1 Bug Fixing

From the evaluation it was evident that everyone believed that the tool would make it easier to understand what had caused the performance issue and thereby to make it easier and faster to resolve it. But they also mentioned that it was dependent on the complexity of the problem, what type of performance bug it is and what information is accessible. Regardless, it was repeatedly mentioned that the information at least would help by indicating were to look for a performance issue and give an overview of the problem. One interesting idea was that it would be useful as a way of learning what parts of the system is prone to performance issues and what parts are sensitive to it.

5.3.2 Issue Detection

All of the developers thought that the tool would help in detecting performance issues, but one developer also mentioned that the tests that the tool rely on do not have total coverage. This means that some functionality would not be measured by using the unit tests. At the same time another developer mentioned that unit tests could produce better results than performance testing, since unit testing test more realistic cases compared to performance testing which tests how far the system can be pushed. One developer also made a distinction between identifying an issue and noticing the issue. The developer argued that the identification process of measuring the execution time of unit tests could potentially produce a lot of false positives and that the developer might not notice an issue. The developer added that an issue might go unnoticed for several reasons: it is not trivial to distinguish what is normal behavior in the system and that a lot of tests can make it hard to get an overview of the test results. One developer mentioned that the tool might help to detect performance

issues in cases, such as making quick patches, were the developer does not expect there to be any change in performance. Overall, the developers all agreed that the tool would make it easier and faster to identify a performance issues, but that it also was dependent on the type of information that was given by the tool.

5.3.3 Feedback and Information

From the evaluations it could be concluded that the tool would improve the feedback for performance issues, since it currently is slow and sometimes left out altogether. But it was also mentioned that the speed of the feedback would be improved as a result of less handovers between different developers. One developer mentioned that feedback about performance issues also would be sped up since only senior developers are experienced enough, today, to know what causes performance issues.

Regarding the information accessible through the tool, it was repeatedly mentioned that the measured stack trace in the tool was very valuable, as it allowed one to easily see what part of the system took the most time to execute. Also, one developer argued that comparable information to what the tool provided could already be gathered, but that it was much more work to do so.

However, noise was a problem that was mentioned several times by multiple developers. One developer argued that the testing should be limited to a part of the system rather than all of it, in order to reduce the amount of unnecessary information. Another developer said that there are tests that run every time a developer pushes code which would work well, but then added that there also were some slow tests that would consistently fail if used.

Tests that consistently fail does not only contribute to making the feedback noisy, but it also increases the risk that failing tests are normalized. It was mentioned by a developer that if a test or check would fail consistently and repeatedly, all the developers would start ignoring it. This was also brought up by another developer who stated that the tool could be useful but there currently are projects with warnings that are ignored.

5.3.4 Traceability

The developers all agreed that the tool would increase the traceability to some degree. One developer specifically mentioned that the tool would make it easier to see who had introduced a performance issue, but added that it would be useful to be able to trace it in both directions. Another developer stated that the traceability would be increased since it would be easier to link a performance issue to a developer.

5.3.5 Closure

The answers regarding closure were all consistent between the different developers with a difference in when they feel closure. Two developers stated that they generally feel a sense of closure when merging their code to the main branch, although one of the developers also mentioned that they feel 99% done after their code as been pushed and reviewed. One developer argued that it depends on their knowledge and that a sense of closure would be felt

earlier if she or he had adequate knowledge about that specific part of the system. The remaining developer said that it depends on how extensive the implementation is and that closure would be sensed earlier when implementing a small patch, compared to a larger feature that might be followed up and improved on for some time. But, the developer also mentioned that a check might give a false sense of safety. However, all of the developers agreed that the tool would probably not lead to an earlier sense of closure, but rather an increased sense of closure.

5.3.6 Disadvantages

Even though the developers were generally positive to the prototype the developers identified some potential disadvantages and problems with the tool. One problem that was repeatedly mentioned was that the tool could have an impact on how long it takes to run the tests on Jenkins and that an significant increase could cause a disruption of the current workflow. The developers did not mention a specific threshold time for when it would be too long, but one developer said that it would be fine as long as it did not cause it to be five times as long as it currently takes. However, as discussed in the previous interviews in 3.3, the interviewees could tolerate waiting up to 15 minutes. This is a subjective matter and the actual time might differ a fair bit between different developers.

Furthermore, it was mentioned that having a set threshold time for when something is considered too slow could introduce false alarms when there actually are no problems. The developer also suggested a potential solution by allowing different threshold times to be set for different parts of the system. Similarly, another developer mentioned the reversed problem, that the developers could get a false sense of security from the check. The same developer also said it could be problematic to run several builds simultaneously as they could affect one another and thereby produce unreliable results.

Another interesting problem that was mentioned as a part of the increased learning that the tool could help to augment was that the tool might become superfluous once the developer has learned enough about the system.

5.3.7 General Discussions

The reaction to using the tool was overall positive, but in addition to the topics that already had been discussed, there were some other thoughts and opinions that the developers brought up. One developer expressed that it would be helpful if the tool could provide additional information about the database queries in the tool, such as if the query was lazy, the query select statement and to see if the query was slow due it not being indexed. The same developer also mentioned that it would beneficial to be able to have a link to the specific pull request from the tool.

Another developer brought up that the tool could be useful for optimizing code. For instance by AB-testing two different ways of completing a task, such that one can see which is faster. The developer also mentioned that this could help in cases where one has to decide between readability and optimization, the example that was brought up was that if both versions of the code were equally fast, the most readable one should be used.

5.4 Requirements Gained from Prototypes

This section will reflect on the previous sections in order to distinguish the additional requirements that are relevant for the requirements specification. This is important as it will help in making the requirements specification more complete.

From the two prototype implementations and prototype evaluation new requirements emerged. The first requirement that could be derived was that the tool should be able to *run on developers machines, which run MacOS.* This requirement is important because if the tool is not compatible with the operating system, developers would not be able to use it and thereby it would be useless. In addition, another requirement that could be derived was that the tool should be able to *run on the build server operating system, which runs on Linux.* As with the previous requirement, it is important that the tool is compatible with the operating system otherwise it would be useless.

Another requirement that could be derived was that the tool should be able to *handle Java execution.* As the system at the case company runs Java, it is crucial that the tool can handle Java. This requirement has the same purpose as the previous requirements where the tool would be useless if it can not handle Java. From the prototype, it was also possible to derive the requirement that the tool needs to be *incorporated in the Maven build file.* Since both the prototype and the system at the case company is built with Maven, we believe that the tool incorporated into Maven should be a requirement.

In the prototype, Docker was used to create a set up that behaved the same for every machine. Because the tool builds on the Docker set up, we chose to add *Docker* as a requirement for the tool. Github was used as the collaborative version control tool in the prototype as well as at the case company and we therefore considered that *Github integration* should be added as a requirement. Without this requirement, the tool would be too different from how the work process at the case company looks today and could be perceived as cumbersome. The last requirement that could be derived from the prototype was that the tool should be able to *run on a Jenkins build server.* We chose to add this requirement as Jenkins is used at the case company to run more comprehensive tests before it goes into production. As the prototype is meant to monitoring tests, we considered that the tool should be able to monitor tests on Jenkins as well.

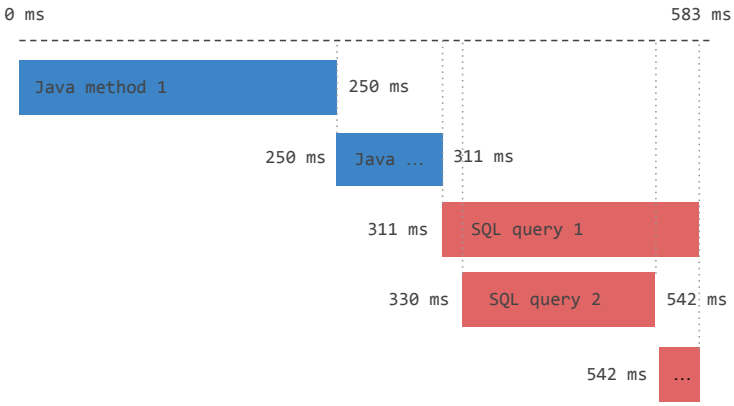

Figure 5.3: The figure shows an example sample of how the timed stack trace should be visualized.

As mentioned during the discussion in the evaluation of the prototype, the visualization of the stack trace was a useful part of the tool. The tool should therefore be able to *visualize the timed stack trace of the system as the figure 5.3.* The stack trace visualization should visualize all the technology layers, for example Java calls and SQL queries as shown in the image.

During our implementation of the prototype we also noticed that it was not entirely easy to configure the monitoring to work with the testing. The tool should therefore be able to be *configured and integrated with TestNG*, which is the testing framework used at the case company.

During the evaluation it was also brought up that the tool should have provided more information about the queries, such as if the query was lazy, to be able to see the query select statement and whether or not it was slow due to it not being indexed. It was also mentioned that the tool should also provide a link back to the pull request, which would also increase the traceability.

The extended requirements specification can be seen in Appendix E.

Chapter 6
Results

As the results have been scattered over previous chapters, the purpose of this chapter is to summarize all the results from these chapters. The results and the final requirements specification will be discussed in order to provide answers to the research questions. This is important as research questions help us answer the initiating problem. In addition to the results on the research questions, four other results emerged: a proof of concept, a validation of the problem, technical experience and a core requirements specification.

The proof of concept emerged as a part of investigating what requirements developers have on the tool, discussed throughout the project. The proof of concept has shown that there is a viable way of handling the problem, even though the solution implemented in the project might not be the ideal solution to the problem.

The validation of the problem has continuously been taking place throughout the project along with the activities carried out, such as during the interviews. It is evident that taking performance into account when developing software is a real issue that has an impact on both the development process itself and the quality of the product.

Also, much experience has been gained from implementing the prototype that would be of use in case an actual product were to be implemented. For example, our experiences from the first prototype lead us to choosing of the shelf components for our solution that shifted the focus from creating a new tool from scratch to integrating several tools with each other such that they would act as a single system.

But, one of the most important results from the projects is the requirements specification, which has been in development during the entire project. The requirements specification concludes our finding about what developers need in a tool to solve the initiating problem but could also acts as a starting point for further research.

The rest of this chapter will address the research questions and then specify the core requirements in the requirements specification. We will begin by discussing the results of the research questions by analyzing the results from the baseline from the questionnaire and the evaluation of the prototype as well as compiling the final requirements specification. This is followed by a requirements specification containing the core requirements that we

considered to be most important.

6.1 Results to Research Questions

In this section, the results for the research questions will be compiled and discussed by summarizing results from previous chapters. This is important as compiling results will help us understand if we were able to solve the initiating problem. This section will begin by discussing **RQ2** as it has played the largest part during the course of the research, followed by **RQ1**, **RQ3** and finally **RQ4**.

6.1.1 Requirement Specification

In this section, the final requirements specification will be discussed in order to compile an answer to **RQ2**, *What is useful information for the developer, what requirements does the developers have on the tool?*.

The final requirement specification could be derived after the evaluation of the prototype and further observations from us. Then both the initial requirement and additional requirements from the design and prototype stage had been included together with our observations during the experiment. In Appendix E, the final requirement specification can be found.

6.1.2 Development Process

This section aims at compiling an answer to **RQ1**, *What effect could an automated tool have on the feedback loop and development process?*, by analyzing answers from questionnaire and evaluation of the prototype. By analyzing the answers, it is possible to come to a conclusion on whether developers experience faster feedback about performance issues.

From section 3.2, we could set the baseline for developers being notified about an issue in their code to once a month, whereas developers being notified about performance issues happens more seldom than once a year.

Measurement parameters	Baseline	After prototype evaluation
Notified about performance issue in code	More seldom than a year	From testing locally or every pull request
Fix someone else's issue	Once every six months	Does not happen since you get feedback in CI
Good traceability within the development process	Agree to neutral	Improved with traceability within stack trace
Easy to link issue in production to code change	Agree	Improved and moved to CI
Easy to link code change to developer	Strongly agree	Unchanged but moved to CI

Table 6.1: Comparison of measurement parameters for the development process between baseline and prototype

After evaluations of the prototype in section 5.3, developers expressed that the prototype would improve the development process and above all improve the aspects of feedback and traceability for performance issues. A comparison between baseline and evaluation of the prototype can be seen in table 6.1.

From our own observations, we were able to compile that the following improvements could be made to the development process. The first improvement was that the prototype provided faster feedback on performance issues by already integrating it in the Continuous

Integration phase. The time a developer has to wait for feedback is the maximum time it takes to build and run tests on a build server. This means that the feedback loop is improved from a week/month to about 1 hour. The second improvement was for traceability and there it has been improved for several aspects. The prototype improved traceability for several aspects. One aspect is that traceability in the development process was improved by going from looking among the majority of developers to a developer to find out who has introduced the performance issue. Furthermore, the prototype has improved traceability in the call stack itself, where the prototype measures the time for all method calls. A third improvement is issue identification where the method calls are measured in time to identify issues and make it easier for the developer to search the code base.

It was also mentioned by the developers that tested prototype in chapter 5 that the tool could change the development process by allowing developers to AB-test different ways of solving a problem, similar to test driven development. One developer also mentioned that the prototype could be introduced as a tool for increasing learning in development process, as new developers without much previous knowledge about the system can get an understanding of what parts are performing well and less well.

6.1.3 Bug Fixing

This section will compile an answer to **RQ3**, *How would the time required for bug fixing be affected with the use of an automated tool?*, by comparing the result from baseline with the result from prototype evaluation.

In section 3.2, the baseline for bug fixing is set to developers discovering bugs during testing and developers had to find the bug without any clear indication where in the code base i exists.

From section 5.3, we could compile that bug fixing could be improved in the sense that the prototype made it easier for developers to understand what had caused the performance issue. Bug fixing could be improved by indicating where in the call stack that the performance issue occurred and thereby shorten the time for bug fixing.

6.1.4 Closure

This section aims at comparing the result from baseline with the result from prototype evaluation to compile an answer to **RQ4**, *How could a shorter and rigorous feedback loop give developers faster closure?*.

From section 3.2, the baseline for closure could be set to when a developer has had his code changes peer reviewed by two other developers and can merge the code changes with the main branch in Github.

After the prototype had been evaluated in section 5.3, we were able to compile that the result for closure was that developers felt closure as quickly as before they tested the prototype, but that the feeling of closure was stronger with the help of the prototype.

6.2 Specifying Core Requirements

This section aims at compiling the core requirements from this research. Core requirements were derived from the different stages of the research and from our own observations. This is important because the core requirements have been validated in the various phases of the research and proved to have a prominent role in interviews, design and prototype. Furthermore, this is important because the core requirements could apply to other context and be a starting point for other's implementation of a tool.

After the requirements specification had been iterated in the various phases of the research, certain requirements appeared to be fundamental for the tool to exist and we considered that these were the core requirements in our research. The core requirements were recurring in each phase of the research and can be seen as a starting point for others who intend to implement their own tool.

In addition to the core requirements, some requirements appeared to be more valuable than others. These requirements could be validated in the different phases of the research by developers repeatedly stating that the tool should contain these requirements and based on our own observations of what was important. During the iterations, it became clear that the most important areas of information were information given to the developer, the tool is part of Continuous Integration and that the tool should be automated. Therefore, the requirements that fit into these areas have been filtered out to form a subgroup in the final requirements specification.

In Appendix F, the final requirements specification can be found with the subgroups core requirements and most valuable.

Chapter 7
Discussion & Related Work

This chapter will reflect on our work performed in this thesis, threats to validity, how generalizable the results are, related work as well as future work. This chapter is important as it discusses the efficiency of the research itself which allows the reader to both understand and learn from the complications experienced during the research. It also investigates how well the results would apply in a general situation such as at another company or workplace. Furthermore, the chapter will discuss related work and how it connects to the results of the thesis. It will also discuss ideas for future work such as how the research could be extended in case the reader is interested. The chapter will start off with a section discussing our reflections on the methodology. The following section will continue the chapter with a discussion of the threats to validity present in the research. The next section will then discuss how generalizable the work is to another context. This will be followed by a section discussing related work and lastly a section about future work.

7.1 Reflections on Methodology

This section will discuss and reflect on the execution of the thesis, what worked well, what could have been done better and what we would like to do differently. This is important for the reader to understand the limitations of the research while also giving an insight into how to conduct similar research.

7.1.1 Qualitative and Quantitative Results

Comparing qualitative and quantitative results is relatively difficult as qualitative results are descriptive compared to quantitative results that are countable and measured in numerically. In the beginning of the project we decided to send out a questionnaire to the DevOps department at the case company in order to set a baseline for how long it took developers to

resolve bugs, when they feel a sense of closure, how fast and frequent feedback about performance is today and the current state of traceability. This baseline was quantitative in the sense that the developers would estimate the answers either in time or by choosing from a 5-point Likert scale. Later in the project, however, we did not measure these topics quantitatively but rather qualitatively with the reason being that the developers had difficulties estimating them. This had the effect that quantitative measurements had to be analyzed and compared to qualitative measurements. The results therefore gave an indication to whether these aspects had been improved or not, without specifying the quantitative difference. If this would be redone, we would settle for either a qualitative or quantitative approach as the measurements would be easier to compare. Also, it is probable that this specific investigation would be better researched qualitatively since the developers thought it difficult to estimate the topics.

7.1.2 Biased Scoping

There is a possibility that the research has been subjected to biased scoping as we intended to do the research in the field of configuration management. That intention may have contributed to us not being open to other solutions that are not within the configuration management area. When we reflected on the problem domain afterwards, we realized that it was more related to testing than we initially thought. The fact that the problem domain is in another area can therefore be seen as a threat to the validity because it can be perceived that we have misunderstood the problem domain from the beginning. We want to argue that the problem domain is still in the field of configuration management, but that testing is included more than we initially thought.

7.1.3 Literature Search

When we searched for literature, we had decided to use academic search sites to find credible sources. However, it was difficult to find research similar ours as we intended to solve the problem in a slightly different way than others, due to . In addition, a more focused scoping would have helped to know more clearly what we were looking for, so we only found a few sources that were relevant. With that, a lot of time was spent searching for literature that could instead be spent researching the problem domain more deeply. It also contributes to the fact that the sources we found do not relate so much to the problem and we had to search for more sources at the end of the research. By finding additional sources at the end of the research we could improve the literature.

7.1.4 Neutral Answers in Questionnaire

Furthermore, the questionnaire had been designed with the use of a 5-point Likert scale as previously mentioned, with the middle answer being neutral. This had the impact that it allowed the developers to not take a stand i the questions and effectively reducing the amount of data available for analysis. If the questionnaire had been redesigned, we would consider using a 4-point or a 6-point Likert scale as it would force the developers to take a stand on the topic, even though their opinion might not be strong.

7.1.5 Hard to Answer Questions

It also came to our attention that the questionnaire was sometimes thought hard to answer by some developers as they felt they did not understand the questions. This occurred even though we had tested the questionnaire on an experienced developer before sending it out. It could be the case that the questions were easily understood by the experienced developer, but not by less experienced developers. This could probably have been remedied by also testing it with a less experienced developer such that we could make sure that it would be easily understood.

7.1.6 More Questions About Closure and Traceability

When we formulated questions for interviews in chapter 3, the main focus was on feedback in the development process, which meant that most of the questions were about feedback in various aspects. Since the focus was on feedback, it meant that traceability and closure ended up a bit glossed over and few questions were formulated about these topics. This may have contributed to traceability and closure not being properly explored and thus requirements may have been ignored. If questions were to be reformulated, we would formulate more questions about traceability and closure to make sure that we have explored those areas properly.

7.1.7 Pre-analysis

The pre-analysis was carried out with the purpose of formulating research question for the initiating problem. The issue that we were investigating seemed fairly simple and focused when we started out the project. Looking back at this phase we now realize that the problem was much wider and complex than we thought, which could evidently be seen in the range of different discussions we had during this phase. The project would have benefited by a more focused scoping earlier in the research process in order to be able to focus our resources on a small part of the complex problem. Furthermore, a more focused scoping could have saved some time that instead could be put on implementing a prototype or explore further designs within the focused area. On the other hand, if we had not had a broad focus area, we would not have been able to explore so broadly and thus have been able to miss potential solutions.

7.1.8 Fewer Design Proposals

There were five design proposals in the design phase of this project. While not being too many to justify, their focus could have been altered a bit. The two first designs were roughly the same, with a few alterations and the last three were also similar with a few alterations. The designs could have been reduced to two more general designs, and allowed us to possibly propose designs that we did not have time to investigate. The granular alterations of the designs could then have been investigated as a separate step to the design process. In the end, the various proposals did not provide enough value to be worth putting so much effort into them.

7.1.9 Requirements Specification

The requirements specification consist of requirements that were identified during the project. However, few resources were put into specifying and detailing the requirements more thoroughly. If more time had been available it would have been beneficial to specify the requirements as use cases that the tool should support. Not only would this allow for a more detailed requirements specification that is easier to understand, but it would also allow multiple requirements to be combined and specified once.

7.2 Validation

This section will reflect on the work and discuss the threats to validity present in the thesis. This is important as it allows the reader to assess the claims and results based on the threats that we have identified during the project.

7.2.1 Sample Size

This section discuss the validity of the sample size from the questionnaire and interviews and what threats there exist against it.

The sample size of both the questionnaire and interviews were limited and only included developers at the case company. This could have an impact on the validity of the results since the developers at the case company might have differing opinions to those working at other companies and since the specific developers that contributed in the interview or questionnaire might have an opinion that does not reflect the general opinion at the case company. While it may not be enough to definitively deliver conclusive results, it is enough to give an indication.

7.2.2 Biased Interviewee Selection

This section discuss how the selection of interviewees could have been biased and how it can be a threat to validity.

Interviewees were interviewed partly based on experience with performance issues at the case company. They were chosen because we considered that we would get the most knowledgeable answers about the problem domain and requirements for tools. There are advantages and disadvantages to choosing these interviewees where an advantage is that clearer requirements for the tool emerge because they have good insight into the problem. One disadvantage is that a broader opinion is missed, a developer with less experience may make other demands on the tool. By having more diversity in terms of experience of the problem domain among the interviewees, other requirements could arise but we consider that we have been able to cover a broad enough range to have covered most parts among developers.

7.2.3 Untested Requirements

This subsection will discuss the lack of tested conclusions. Some of the subconclusions and conclusions are not tested and as such their validity could be argued.

The requirements that emerged from interviews included a wide range of the development process, which made it difficult to validate all the requirements that arose. Through the design and prototype phases, we only had the opportunity to validate parts of all the requirements. Our opinion is that the requirements that have not been validated are still relevant to solve the problem, but that these requirements would benefit from being validated once more in case they are to be used as a basis for further research or implementation.

7.2.4 Testing the Prototype

This subsection will discuss the threat to validity in regards to the tool never being evaluated or tested as a part of the actual software development process.

The tool was evaluated by developers by exposing them to a scenario in which they had to uitilize the tool in order to find an error in the code. While this setup can give an indication of the experience of using such a tool in a development process, it is not enough to find conclusive results. It is possible that, would the tool be tested in an actual software development process some requirements would have been redundant or undesirable and others needed. It could also have another impact on the software development process than what the developers thought it would have. It is also possible that the tool was perceived as overly effective as a results of it being evaluated in a scenario in which we had a high degree of control over the variables that could effect the tools efficiency.

7.3 Generalizability

This section will discuss and reflect on the results of this research and how it adopts to other companies. This is important as it shows that this problem can arise in other development environments and the degree to which the solution in this research can be adopted by other companies.

Performance is an important part of a product or system and many companies place great emphasis on performance when developing their products. Therefore, the problem domain in this thesis can be found in many other development environments where agile methods and a Continuous Integration/Continuous Delivery process are implemented. No company wants performance issues to be released to the customer and therefore it is advantageous to be able to test the product/system before. One possible difference from other companies is that the case company delivers a service where the user experience is important, which in extension means that satisfactory performance is important. There might be some contexts or companies in which the performance of the service is not as important and were it might not be an issue. But in general, the results of the project is generalizable in this aspect.

In software development processes, feedback is an important component for developers to know if they have achieved the desired behavior. The feedback area can relate to other companies and organizations when you want as much feedback as possible to be able to make good decisions, where performance is part of the feedback. The case company uses an agile software development process which emphasizes the focus on feedback during the development process. This is in contrast to more waterfall-like projects or companies were less emphasis is put on feedback. Therefore, the results for feedback in this project are mostly generalizable, depending on the software development process used.

Another component that is relevant to other companies is traceability. By being able to more easily troubleshoot where a performance issue has arisen, traceability increases. Both in terms of the development process itself but also in the code base itself. As the saying goes, time is money. And what company does not want to be able to save money by streamlining the development process. It is however important to note that the need and use cases for traceability might differ between different contexts. For instance, the case company delivers software as a service in which bugs and issues can be fixed retroactively relatively easily. In a context where a system is delivered preinstalled on a chip, traceability might play a different role. In that very context it might be very useful to be able to trace bugs and issues in code to a certain batch of chips, such that these chips can be sent back and be reprogrammed. While traceability might in general be beneficial its uses might differ between different contexts and therefore the results for traceability in this project is generalizable to most contexts that delivers software as a service.

Therefore, we think that the requirements specification developed in this research can be seen as a starting point for other companies that want to develop their own tool for testing and detecting performance issues. The requirements that have been validated can be considered as a good basis for the tool, while those that have not been validated should be adopted with caution.

The prototype developed in the research could also be relevant to implement for other development environments similar to the case company. That is, you connect an agent on the code base and then run tests, either locally or on a build server.

7.4 Related Work

This section will reflect on related work to this thesis. The related work will be discussed on different approaches to solve a similar problem and why it could or could not work for this thesis.

7.4.1 Closing the Feedback Loop in DevOps Through Autonomous Monitors in Operations

Hrusto et al. [9] investigates the problem of relaying feedback autonomously in a DevOps environemt. This article is interesting as it investigates problems that were also discussed in this thesis, such as how to make sure the feedback is not buried by noise, how to direct feedback to the correct developer as well as knowing what feedback to actually send. This article thus gives insight into problems about how feedback processed and sent, which is important when designing a feedback loop.

Summary

Monitoring a system in production is an important activity to be able to ensure the system is working as intended and often times developers and operators set up alerts to notify them whenever anything goes awry. The article investigates how to improve the feedback from operations in a Swedish company responsible for managing tickets and conceptualize the problem as three separate problems: not notifying the corrects person or team, sending so many

notifications that they flood the channel and bury important notifications and lastly that the system might send notifications as a result of errors in external systems. Their contribution is three fold, they provide a problem conceptualization, a solution design and a prototype. In order to obtain their results they first conducted six interviews with senior employees with different responsibilities from where they formulated the conceptualized problems and thereby the solution design that laid the foundation for the prototype implementation. The result of their study was conceptualizing the problem with operations feedback, designing a solution for how the conceptualized problems could be remedied and lastly the implementation of their design solution using machine learning to create a smart filter for automatic error detection. In a series of tests and real usage their solution performed well, even notifying an error 30 minutes before it was reported by an user.

Discussion

In the article there is an interesting discussion about ”alert flooding” and noise in regards to frequently sent notifications. The authors refer to this noise as notifications sent mostly due to internal or external temporary glitches and then proceeds to propose a solution that aim to filter these out. This is also relevant to the context of this thesis, since the feedback that is being relayed to the developer is also subject to alert flooding. It is therefore important to make sure that the feedback that is delivered to the developers is concise and does not include information that dilutes the feedback and makes it less actionable. Furthermore, alerts or feedback can become unmanageable when they are not directed to a single person, as discovered in the article. This is also a problem that is relevant to the context of the thesis. The approach ultimately used in this project is that the developer who is committing code is responsible for making sure that the code is working as intended and therefore are any issues connected with the commit directed to the developer. However would the performance testing be introduced as a discrete quality check before releasing the code into production or monitoring the code in production it would not be as easy. Then it would be needed to analyze the code to be able to direct the feedback more accurately to a developer. In the article machine learning is used in order to create a smart filter that can learn what alerts to send and which to suppress, which could possibly also be used in the context of this thesis.

7.4.2 Shortening Feedback Time in Continuous Integration Environment in Large-Scale Embedded Software Development with Test Selection

Koivuniemi [10] explored how much testing time could be reduced while maintaining fault finding capability using test case selection. We found it relevant to our research as Koivuniemi chose test cases through which files were modified by the developer and in our research we intended to test only one developer’s changes but to be able to eliminate noice.

Summary

In continuous integration, one of the main principles is to integrate often into the mainline. As the code base grows so does the test suite, which means that integration takes longer

time. Integration time taking to long, might make developers reluctant to integrate often and thereby feedback time gets slow. Therefore, it is necessary to investigate the possibilities to find a subset, test case selection, of the test suite through automation to shorten the feedback time of testing in Continuous Integration and still being able to have a stable product.

This master's thesis provides a designed artifact to enable automated test case selection in large scale software systems, such as embedded systems, to cut feedback time and thereby enable continuous integration. Furthermore, the paper contributes with as extended knowledge base to practice continuous integration in large enterprises as well as offering a solution on how to scale continuous integration in embedded software development.

The methodology of this thesis is based on Design-Science Research which contains seven guidelines of which some were selected for the thesis. The first step takes on the design of an artifact and in this thesis it is the design of a test case selection algorithm. In the next step research of system takes place and to look at the gap between the goal state and the current state of the system, to decide the problem relevance. Then, an evaluation of the artifact is performed through well-executed evaluation methods, where the main area of evaluation is feedback and the difference between the test selection tool and current system. Furthermore, it was evaluated if the test selection tool catch the same faults as the re-test all technique. Following evaluation is the presentation of contributions through the design artifact and extension of the knowledge base within the practice of continuous integration in large enterprises. During the next step, the design is improved by iterative cycles with input from developers at the case company. At last, the results were presented together with implications for the whole product.

The paper came to the conclusion that it is possible to safely shorten the feedback time by 55.7 % in 404 commits. It was found that six faults slipped through the tool in 278 commits and as these faults were caused by changes to the test code it was well-acknowledged that they would slip through. Hence, the tool could be accepted as relatively safe. In addition, it was stated that continuous integration is one of the most important things affecting successful adaptation of agile software development at scale.

Discussion

To begin with, we deem that it is a well-conducted research where Koivuniemi has chosen to validate his thesis by testing two different test suites on different teams. Therefore, we believe that the research can be assessed as valid. However, the methodology for this research is difficult to follow as it is not clearly described in which order the various phases have been performed, but we believe that it does not affect the results too much.

From this research, we got inspiration for how the tool can filter out the changes that developers have made to the code base. For our purpose, it was to distinguish between a developer's changes to the codebase and the rest of the codebase to know which tests to monitor. While the purpose of the article was to filter out a smaller test suite to be able to reduce the time to run tests. Since the method for test case selection in the article has high results for fault finding capability, we considered that the method would fit well in this research even if our focus is a little different. Another contributing factor to why we drew inspiration from this article was that both the article and our case company are in the same area, embedded systems. Due to lack of time, we did not have time to integrate this method for test case selection in our prototype and this is something that could be seen as future

work.

One part that differs between the article and our research is the problem analysis. In the article, it is poorly described how the problem analysis has been performed, which as a reader makes it difficult to understand how the article has analyzed the problem. As we interpret it, the problem analysis has been based on observations, just as in this thesis, but could have benefited from, for example, interviews also being held. By using several different ways to analyze the problem, you get a more nuanced picture. It may be that the article used interviews but nothing that we perceived when we read it and something that we would like to discuss with the author.

7.4.3 Unit Testing Performance in Java Projects: Are We There Yet?

Stefan et al. [15] investigated the usage of unit performance testing and its usage in open source projects on GitHub. We found this article interesting as it researches how much unit testing is used for performance testing, what kinds of projects that use it, how long it takes and if it can detect changes in performance but also investigate how much performance measurements change in existing projects. This could give us insight into whether or not using unit testing for performance testing is a efficient approach to testing performance.

Summary

There are two reasons to why the research is important: Performance testing is an acknowledged method of quality testing, that is often deemed the culprit of many failed projects. It is also recognized that early testing is cheaper than testing late. The research analyzes 99019 different open source projects that uses performance testing. They conducted a survey with 111 developers that had contributed projects using the performance testing framework JMH. They identify what projects are suitable for unit performance testing. They explain what changes they make to their performance testing framework, SPL, that they presented previously in another article. They analyzed the 99019 different open source projects by searching for known code patterns that indicates the usage of a specific performance testing framework, followed by them trying to execute the identified tests. They then ran these tests at different times in the repository's history in order to investigate the performance change over time. They then categorized the repositories that measured performance in order to categorize what projects are more likely to measure performance. Then they measured how long it took to execute the performance tests in order to investigate how long the tests take to run. They also investigated whether or not unit testing of performance actually reveal any changes in performance by running the performance unit testing before and after a change. Lastly, they sent out a survey to developers that had participated in projects that used the performance testing framework JMH.

Their conclusions were that only 0.37% of all the projects actually use a performance testing framework and only 62% of them produced a result within 4 hours. It was also concluded that, projects using JMH, performance tests are usually introduced later in the lifetime compared to functional unit testing. From the surveys they concluded that automation of performance testing is an issue with 77% responding they carry out performance testing manually,

61% automated evaluation and 50% build integration. It was also concluded from the interviews that 47% preferring performance testing at commit and that 37% preferring running it before release. From the survey they also concluded that 23% of developers acted on performance regression or improvements and 57% that used it when making design decisions.

Discussion

Even though performance testing is considered an important part of quality assurance and it often being the cause of a failed project it is interesting that the majority of performance testing relies on manual testing rather than automated performance testing. Testing performance with unit testing could potentially yield the same benefits that functional unit testing does, such as fast, consistent and automated testing that could be used as a part of development rather than a separate following step. The consistency with which the testing is carried out could allow performance regression to be detected before the code is released into production and thereby reducing the risk of the system performance to regress over time. Furthermore, it could also have an impact on the development process itself, by emphasizing the focus on performance during development. This could have the effect of developers considering adequate performance equally to correct functionality. As also mentioned during the evaluation of the prototype in 5.3.7, this could help with comparing solution to one another and make better design decisions. It is also interesting as the article investigates the use of unit testing in order to test performance, which is relevant to the context of this thesis since it is also based on unit testing. It also indicates that it is possible to extend the regular functional unit testing by also using it to test the performance of a system.

7.4.4 On Agile Performance Requirements Specification and Testing

Ho et al. [8] proposes a model for how to produce a performance requirements specification in agile development environments. We found that this article could be relevant to our research as the case company implements an agile development process where the model could be used to develop performance requirements that can then be implemented in the tool.

Summary

In agile software development, the authors have identified a lack of practices to specify performance requirements and how to collect a requirements specification regarding performance which can cause performance issues in a software system. Due to agile processes, it is difficult to make a preliminary analysis of performance requirements and something that is not desirable. Since performance is not specified in the beginning, it will be easy to forget it during the development. Therefore, the authors considered it necessary to develop a model for developing and specifying performance requirements that can be used in agile development environments.

The paper proposes the Performance Requirements Evolution Model (PREM) which serves as a guideline for performance requirements and validation. PREM divides performance requirements into four levels where the first level represents a qualitative, casual description. In the second level, the requirements are specified more by adding quantitative

measurement values to the description. In the third level, the requirements are specified as quantitative performance objectives with system execution characteristics of the system. In the last level, the performance requirement from the actual use of the system is specified. Based on the model, developers can then write performance tests and as they get into the development of the system or feature, they can specify the performance test more. The model was then validated in a software project that had a high focus on performance where the results showed that performance had an upward trend and that it fits well into agile methods.

PREM is based on agile principles such as stories and test automation in Extreme Programming but also on principles for performance testing. The authors begin by providing the reader with an overview of PREM and how PREM fits into agile methods. To validate their theory and model, the model is implemented in a software team that develops performance-critical software for hardware components. The team started by specifying resolved requirements with the help of customers, in accordance with the first level of PREM. Subsequently, a domain expert specified performance measurement values for the software in accordance with level two of PREM. In accordance with level three in PREM, the software was tested on several hardware components simultaneously for quantitative performance objectives with system execution characteristics.

The conclusions that can be drawn from this paper are that it contributes with PREM, performance requirements evolution model. It can be used as a guideline to gradually identify and specify performance requirements in an agile development environment. Furthermore, developers were able to write appropriate tests using the model.

Discussion

With the help of the model proposed in the article, agile work processes can be improved with respect to performance. By testing the model in a software team and evaluating it with the help of the team, we believe that the statements are valid. With the help of unbiased people, the fair evaluation could be done. Since the model has been tested in a software team where performance is a high priority, it makes us wonder how the model had performed in a software team where performance has a lower priority. The article mentions that teams can be tempted to start at a higher level of performance requirements, which could lead to hasty decisions for performance. This could be the case with the case company where performance is a lower priority compared to functionality and that one therefore speeds up the specification process of performance requirements. Since you do not have time to give the performance requirements the reflection that is needed, it can lead to you producing poorer performance requirements and therefore need to go back to re-evaluate and rewrite tests.

The article relates to this thesis in that the performance requirements evolution model (PREM) can be seen as a prior step to our tool. With the help of PREM, the developers at the case company can develop clear requirements for the performance of their systems. Because all developers know what performance requirements are placed on the software, they have clearer goals to strive for when they develop and test their code, which hopefully results in fewer performance issues in the code base.

7.5 Future Work

In this section, the authors will suggest future work based on this thesis that has emerged during the performance of the thesis. This could give readers ideas to further investigate and explore the area for research.

One interesting discussion we had around testing performance was the reliability of the results. Each time a method is monitored and logged by the monitoring software it gathers a sample of how long time the method takes to execute and more samples generally provide a better and more reliable result. But as mentioned this requires methods to run multiple times. It would therefore be interesting to investigate whether the unit tests can be run multiple times such that a certain degree of statistical significance is achieved.

Unit tests were in this project used as a method for generating a load on the system such that the execution time could be measured. It would however be interesting to investigate how this load could be generated differently, possibly by capturing user interactions and replaying them as a way to generate a load. It could also be achieved by writing tests that are specifically designed for testing performance. This could possibly have the effect of making the test results more reliable. This also opens up new ways of validating the performance, such as writing tests that are also validated similarly to unit tests, for example by setting execution time limits or timeouts.

While this thesis foremost investigate the ways in which performance issues can be identified during testing, it is only one method of identifying performance issues. It is also possible to, in conjunction with performance testing, monitor the production environment and automatically generate new task issues that are automatically routed to the correct developer. There are most likely many different solutions to such a problem, but automatically routing issues could possibly be done using machine learning similar to the article by Hrusto et al. [9] mentioned previously in the report.

It would also be interesting to investigate how the performance of a system is dependent on the hardware configuration. It could for example be in order to run the same testing on hardware that is very similar to the production environment such that the results might would reflect the actual effect on the production system. This could also be combined with machine learning that could forecast how the system would perform in production given the performance on another computer, for instance the developer computer. This could possibly allow the developers to get an understanding of how well their code would work in production without having to actually release it into production.

Chapter 8
Conclusions

The goal of this thesis was to improve the feedback loop and traceability in the development process to reduce the occurrence of performance issues in production. This was done by investigating how to efficiently provide feedback about performance issues to the developers, by deriving a requirements specification based on literature, interviews and observations. The requirements specification then served as the basis for designing and implementing a prototype, which was evaluated by developers.

The initiating problem was the lack of a defined practice in the development process to identify performance issues. This led to performance issues being pushed into production. If performance issues are not detected, they can in the long run also affect the user experience of the service. The problem could be characterized as a lack of traceability between performance issues and the code, which results in feedback that was hard to take action upon. Furthermore, the process of discovering performance issues was not automated, which caused the feedback loop to become unnecessarily slow. Lastly, this could affect the developers' sense of closure since it is difficult to know if the code is adequate.

As a result, a requirements specification was elicited which outlines how the tool would be implemented in a general software development environment. With help of the prototype, traceability was increased between performance issues and the code base. This gave developers more feedback into what had caused the performance degradation. Furthermore, the implementation of a new automatic process in the testing stage allowed the developer to receive more timely feedback. The conclusion could also be drawn that earlier and more actionable feedback did not make developers feel a sense of closure earlier, but rather give them an increased sense of confidence that their code is working as intended.

The requirements specification outlines a tool that can help identify performance issues by increasing the traceability while providing more timely feedback to the developers. Therefore our solution allows for performance issues to be discovered and resolved before they have been released into production. This provides the developers with an extra assurance that their code maintains adequate quality and thereby contributes to a better user experience that is less likely to have performance issues.

References

[1] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[2] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[3] Lars Bendix. A short introduction to software configuration management, lecture note in software configuration management, version 0.80, 2019.

[4] Markus Borg, Leif Jonsson, Emelie Engström, Béla Bartalos, and Attila Szabo. Adopting automated bug assignment in practice–a registered report of an industrial case study. *arXiv preprint arXiv:2109.13635*, 2021.

[5] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE software*, 32(2):50–54, 2015.

[6] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[7] Fergus Hewson, Jens Dietrich, and Stephen Marsland. Performance regression testing on the java virtual machine using statistical test oracles. In *2015 24th Australasian Software Engineering Conference*, pages 18–27. IEEE, 2015.

[8] Chih-Wei Ho, Michael J Johnson, Laurie Williams, and E Michael Maximilien. On agile performance requirements specification and testing. In *AGILE 2006 (AGILE'06)*, pages 6–pp. IEEE, 2006.

[9] Adha Hrusto, Per Runeson, and Emelie Engström. Closing the feedback loop in devops through autonomous monitors in operations. *SN Computer Science*, 2(6):1–14, 2021.

[10] Jarmo Koivuniemi. Shortening feedback time in continuous integration environment in large-scale embedded software development with test selection. *University of Oulu repository*, pages 16–18, 2017.

[11] Oliver Krancher, Pascal Luther, and Marc Jost. Key affordances of platform-as-a-service: self-organization and continuous feedback. *Journal of Management Information Systems*, 35(3):776–812, 2018.

[12] Soren Lauesen. *Software requirements: styles and techniques*. Pearson Education, 2002.

[13] Ian Molyneaux. *The art of application performance testing: from strategy to tools*. " O'Reilly Media, Inc.", 2014.

[14] Mary Poppendieck and Tom Poppendieck. *Lean software development :*. Addison-Wesley„ Boston :, cop. 2003.

[15] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. Unit testing performance in java projects: Are we there yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 401–412, 2017.

[16] Donna M Webster and Arie W Kruglanski. Individual differences in need for cognitive closure. *Journal of personality and social psychology*, 67(6):1049, 1994.

[17] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.

Appendices

Appendix A
Pre-analysis Interviews

Inledning

- Vad heter du?
- Hur gammal är du?
- Har du någon tidigare utbildning?
- Vad är dina arbetsuppgifter på Telavox?
- Har du jobbat med slow queries tidigare? Slow query taskforce? *For context, slow query taskforce is a group of developers that have volunteered to solving slow queries, at the company.*

Huvudfrågor

- Vad är problemet med slow queries?
- Hur ser arbetsprocessen och systemet nuvarande ut för problem? Manuellt? Automatiskt?
- Använder Telavox continuous delivery eller continuous deployment?
- Hur påverkar det arbetet?
- Vad hade underlättat arbetsflödet?
- Vad ser du som en potentiell lösning?
- Hur får du feedback? Får du feedback? Hur lång tid tar det?
- Hur återkopplar ni den här informationen till utvecklaren? Gör ni det?
- Hur vet du om det du pushar upp är tillräckligt bra?

- Hur ser datan ut från produktionsdatabasen?
- Vad tror du är användbar information i datan?
- Hade man kunnat testa slow queries innan produktion?
- Tror du det är möjligt att få ut liknande data innan ändringen är i produktion?
- Är det något mer du vill tillägga?

Appendix B
Problem Analysis Interviews

Inledning

- När är du född?
- Har du någon tidigare utbildning?
- Hur länge har du jobbat på Telavox?
- Vad är dina arbetsuppgifter på Telavox?

Huvudfrågor

- Skulle du vilja ha ett verktyg som ger snabb feedback men inte helt säkert att det fångar en slow query, t.ex. i testningsstadiet, eller ett verktyg som tar lite längre tid men med större säkerhet fångar slow queries, t.ex. från produktion?
- Vilka funktioner skulle du vilja ha i ett sådant verktyg?
- Vilken information skulle du vilja få från ett sådant verktyg?
- Var i CI/CD pipeline hade du velat få feedback på om din query är långsam?
- Var i CI/CD pipeline skulle du vilja att verktyget fanns?
- Hur mycket tid är du villig att lägga på att testa queries i din kod?
- Hur lång tid kan du tänka dig att vänta på feedback om att din query är långsam?
- Hur ofta använder du QueryProfiler? Om de vet vad verktyget är.
- Hur ofta skriver du test för dina implementeringar?
- Hade du föredragit att testa för slow queries i ett GUI eller i kod, t.ex. jUnit?
- Hur och var hade du föredragit att få feedback om att en query är långsam?

Appendix C
Initial Requirements Specification

C.1 Shared Requirements

QuGo1 Verktyget skall minska felsökningstiden med ___%

FuDo4 Man skall kunna specificera vad en slow query är

FuDo8 Verktyget skall kunna visa vilken del av databasen en slow query påverkar

QuDo12 Verktyget skall vara automatiserat

FuPr16 Verktyget skall skicka en notis ifall en query är problematisk

FuPr18 Man skall kunna definera tabeller i en databas som inte får låsas

FuPr19 Verktyget skall kunna identifiera vilket steg i tech-stacken som är långsamt

FuPr20 Verktyget skall kunna ge information om vilka tabeller och fält som orsakar en slow query

FuPr21 Verktyget skall kunna notifiera om en specifik plats i koden som är långsam

FuPr22 Verktyget skall kunna identifiera lösningar till långsamma queries

QuPr23 Verktyget skall inte visa hela sidor med information

QuGo27 Verktyget får inte belasta systemet ytterligare

FuPr28 Man skall kunna ställa in gränser för vad som räknas som en slow query

FuPr30 Verktyget skall kunna skilja mellan långsamma queries och många jämlöpande queries

FuPr32 Verktyget skall kunna summera totala tiden för återkommande queries

FuPr33 Verktyget skall kunna ge standardavvikelse för tiden en query tar

FuPr34 Verktyget skall kunna ge den snabbaste tiden en query tar

FuPr35 Verktyget skall kunna ge den långsammaste tiden en query tar

QuDo38 Verktyget skall kunna köras både i test-stadiet och i produktion

QuPr39 Verktyget skall ta ca.1 minut att använda

QuDo41 Verktyget skall kunna konfigureras i kod

FuPr45 Användaren skall kunna filtrera bort för stunden orelevanta queries

FuPr46 Man ska kunna välja bort de versioner som är testade och inte innehåller långsamma queries

FuPr47 Verktyget skall ha möjlighet att ignorera en viss version av query genom opt-out

QuPr50 Verktyget skall filtrera bort onödigt brus från icke aktuella queries

C.2 Testing (Locally and in Jenkins)

QuDo6 Verktyget skall kunna ge feedback från testning inom 30 minuter om det körs på Jenkins

FuPr9 Vertyget skall kunna ge information om en slow query i GitHub

FuPr11 Vertyget skall kunna ge information om en slow query i Jenkins

QuDo13 Verktyget skall vara ett byggsteg i Jenkins

FuDo14 Verktyget skall kunna identifiera långsamma queries i testning

FuDo15 Verktyget skall kunna varna om långsamma queries i testning

QuGo17 Verktyget skall kunna ge feedback från integrationstester på Jenkins inom ___ minuter

QuPr24 Verktyget skall kunna användas för enskilda commits

QuPr25 Verktyget skall kunna köras lokalt

FuDo26 Verktyget skall kunna integreras som en check i GitHub

FuPr29 Verktyget skall kunna stresstesta koden

FuPr31 Verktyget skall kunna load-testa

QuPr40 Verktyget skall kunna ge feedback inom ca.1 minut om det körs lokalt

FuPr43 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen i Jenkins

FuPr44 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen lokalt

QuPr48 Verktyget ska inte förlänga testtiden med mer än 10% när det körs lokalt

C.3 Monitoring

QuGo2 Verktyget skall minska tiden det tar att åtgärda problem i produktion med ___%.

FuDo3 Man skall kunna välja vilka slow queries som skall monitoreras

QuDo5 Verktyget skall kunna ge feedback från monitorering inom 24 timmar

FuDo7 Man skall kunna välja vilka tabeller som skall monitoreras

FuPr36 Verktyget skall kunna ge queries/minute under en viss tidsperiod

FuPr37 Verktyget skall kunna generera Jira issues automatiskt om en slow query upptäcks

FuPr42 Automatiskt genererade Jiror skall tilldelas personen som orsakade slow queryn

FuPr49 Verktyget skall kunna skilja användaqueries mot interna queries

C.4 Feedback

FuPr10 Vertyget skall kunna ge information om en slow query i Jira om det upptäcks i monitorering

Appendix D
Requirements Specification for Design

QuGo1 Verktyget skall minska felsökningstiden med ___%.

QuGo2 Verktyget skall minska tiden det tar att åtgärda problem i produktion med ___%.

FuDo3 Man skall kunna välja vilka slow queries som skall monitoreras

FuDo4 Man skall kunna specificera vad en slow query är

QuDo5 Verktyget skall kunna ge feedback från monitorering inom 24 timmar

QuDo6 Verktyget skall kunna ge feedback från testning inom 30 minuter om det körs på Jenkins

FuDo7 Man skall kunna välja vilka tabeller som skall monitoreras

FuDo8 Verktyget skall kunna visa vilken del av databasen en slow query påverkar enligt krav **FuPr19** och **FuPr20**

FuPr9 Vertyget skall kunna ge information om en slow query i GitHub

FuPr10 Vertyget skall kunna ge information om en slow query i Jira

FuPr11 Vertyget skall kunna ge information om en slow query i Jenkins

QuDo12 Verktyget skall vara automatiserat

QuDo13 Verktyget skall vara ett byggsteg i Jenkins

FuDo14 Verktyget skall kunna identifiera långsamma queries i testning

FuDo15 Verktyget skall kunna varna om långsamma queries i testning

FuPr16 Verktyget skall skicka en notis ifall en query är problematisk

---

QuGo17 Verktyget skall kunna ge feedback från byggstadiet inom ___ minuter

FuPr18 Man skall kunna definera tabeller i en databas som inte får låsas

FuPr19 Verktyget skall kunna identifiera vilket steg i tech-stacken som är långsamt

FuPr20 Verktyget skall kunna ge information om vilka tabeller och fält som orsakar en slow query

FuPr21 Verktyget skall kunna notifiera om en specifik plats i koden som är långsam

FuPr22 Verktyget skall kunna identifiera lösningar till långsamma queries

QuPr23 Verktyget skall inte visa hela sidor med information

QuPr24 Verktyget skall kunna användas för enskilda commits

QuPr25 Verktyget skall kunna köras lokalt

FuDo26 Verktyget skall kunna göra och visa en check i GitHub pull request

QuGo27 Verktyget får inte belasta systemet ytterligare

FuPr28 Man skall kunna ställa in gränser för vad som räknas som långsamt

FuPr29 Verktyget skall kunna stresstesta koden

FuPr30 Verktyget skall kunna skilja mellan långsamma queries och många jämlöpande queries

FuPr31 Verktyget skall kunna load-testa

FuPr32 Verktyget skall kunna summera totala tiden för återkommande queries

FuPr33 Verktyget skall kunna ge standardavvikelse för tiden en query tar

FuPr34 Verktyget skall kunna ge den snabbaste tiden en query tar

FuPr35 Verktyget skall kunna ge den långsammaste tiden en query tar

FuPr36 Verktyget skall kunna ge queries/minute under en viss tidsperiod

FuPr37 Verktyget skall kunna generera Jira issues om en slow query automatiskt

QuDo38 Verktyget skall kunna köras både i test-stadiet och i produktion

QuPr39 Verktyget skall ta ca. 1 minut att använda

QuPr40 Verktyget skall kunna ge feedback inom ca.1 minut om det körs lokalt

QuDo41 Verktyget skall kunna konfigureras i kod

FuPr42 Automatiskt genererade Jiror skall tilldelas personen som orsakade slow queryn

FuPr43 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen i Jenkins

FuPr44 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen lokalt

FuPr45 Användaren skall kunna filtrera bort för stunden orelevanta queries

FuPr46 Man ska kunna välja bort de versioner som är testade och inte innehåller på långsamma queries

FuPr47 Verktyget skall ha möjlighet att ignorera en viss version av query genom opt-out

QuPr48 Verktyget ska inte förlänga testtiden med mer än 10% när det körs lokalt

FuPr49 Verktyget skall kunna skilja användaqueries mot interna queries

QuPr50 Verktyget skall filtrera bort onödigt brus

QuPr51 Verktyget skall endast skapa en Jira per performance issue

QuDo52 Verktyget skall kunna integreras med Jira

FuDo53 Verktyget skall kunna monitorera produktionsmiljön

QuDo54 Performance tester ska kunna skrivas med ett standardramverk, t.ex. TestNG eller JUnit

FuPr55 Verktyget skall kunna mäta tider i callstacken

FuDo57 Verktyget skall automatiskt kunna identifiera vilka tester som ska köras

QuDo56 Testningen skall vara tillräckligt snabb för att kunna användas i Test Driven Development

FuDo58 Utvecklare skall kunna specifiera vilka tester som ska köras

Appendix E
Requirements Specification for Prototype

QuGo1 Verktyget skall minska felsökningstiden med ___%.

QuGo2 Verktyget skall minska tiden det tar att åtgärda problem i produktion med ___%.

FuDo3 Man skall kunna välja vilka slow queries som skall monitoreras

FuDo4 Man skall kunna specificera vad en slow query är

QuDo5 Verktyget skall kunna ge feedback från monitorering inom 24 timmar

QuDo6 Verktyget skall kunna ge feedback från testning inom 30 minuter om det körs på Jenkins

FuDo7 Man skall kunna välja vilka tabeller som skall monitoreras

FuDo8 Verktyget skall kunna visa vilken del av databasen en slow query påverkar enligt krav **FuPr19** och **FuPr20**

FuPr9 Vertyget skall kunna ge information om en slow query i GitHub

FuPr10 Vertyget skall kunna ge information om en slow query i Jira

FuPr11 Vertyget skall kunna ge information om en slow query i Jenkins

QuDo12 Verktyget skall vara automatiserat

QuDo13 Verktyget skall vara ett byggsteg i Jenkins

FuDo14 Verktyget skall kunna identifiera långsamma queries i testning

FuDo15 Verktyget skall kunna varna om långsamma queries i testning

FuPr16 Verktyget skall skicka en notis ifall en query är problematisk

QuGo17 Verktyget skall kunna ge feedback från byggstadiet inom ___ minuter

FuPr18 Man skall kunna definera tabeller i en databas som inte får låsas

FuPr19 Verktyget skall kunna identifiera vilket steg i tech-stacken som är långsamt

FuPr20 Verktyget skall kunna ge information om vilka tabeller och fält som orsakar en slow query

FuPr21 Verktyget skall kunna notifiera om en specifik plats i koden som är långsam

FuPr22 Verktyget skall kunna identifiera lösningar till långsamma queries

QuPr23 Verktyget skall inte visa hela sidor med information

QuPr24 Verktyget skall kunna användas för enskilda commits

QuPr25 Verktyget skall kunna köras lokalt

FuDo26 Verktyget skall kunna göra och visa en check i GitHub pull request

QuGo27 Verktyget får inte belasta systemet ytterligare

FuPr28 Man skall kunna ställa in gränser för vad som räknas som långsamt

FuPr29 Verktyget skall kunna stresstesta koden

FuPr30 Verktyget skall kunna skilja mellan långsamma queries och många jämlöpande queries

FuPr31 Verktyget skall kunna load-testa

FuPr32 Verktyget skall kunna summera totala tiden för återkommande queries

FuPr33 Verktyget skall kunna ge standardavvikelse för tiden en query tar

FuPr34 Verktyget skall kunna ge den snabbaste tiden en query tar

FuPr35 Verktyget skall kunna ge den långsammaste tiden en query tar

FuPr36 Verktyget skall kunna ge queries/minute under en viss tidsperiod

FuPr37 Verktyget skall kunna generera Jira issues om en slow query automatiskt

QuDo38 Verktyget skall kunna köras både i test-stadiet och i produktion

QuPr39 Verktyget skall ta ca. 1 minut att använda

QuPr40 Verktyget skall kunna ge feedback inom ca.1 minut om det körs lokalt

QuDo41 Verktyget skall kunna konfigureras i kod

FuPr42 Automatiskt genererade Jiror skall tilldelas personen som orsakade slow queryn

FuPr43 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen i Jenkins

FuPr44 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen lokalt

FuPr45 Användaren skall kunna filtrera bort för stunden orelevanta queries

FuPr46 Man ska kunna välja bort de versioner som är testade och inte innehåller på långsamma queries

FuPr47 Verktyget skall ha möjlighet att ignorera en viss version av query genom opt-out

QuPr48 Verktyget ska inte förlänga testtiden med mer än 10% när det körs lokalt

FuPr49 Verktyget skall kunna skilja användaqueries mot interna queries

QuPr50 Verktyget skall filtrera bort onödigt brus

QuPr51 Verktyget skall endast skapa en Jira per performance issue

QuDo52 Verktyget skall kunna integreras med Jira

FuDo53 Verktyget skall kunna monitorera produktionsmiljön

QuDo54 Performance tester ska kunna skrivas med ett standardramverk, t.ex. TestNG eller JUnit

FuPr55 Verktyget skall kunna mäta tider i callstacken

QuDo56 Testningen skall vara tillräckligt snabb för att kunna användas i Test Driven Development

FuDo57 Verktyget skall automatiskt kunna identifiera vilka tester som ska köras

FuDo58 Utvecklare skall kunna specifiera vilka tester som ska köras

QuDo59 Verktyget skall kunna köras på MacOS

QuDo60 Verktyget skall kunna köras på Linux

QuDo61 Verktyget skall kunna användas på Javakod

QuDo62 Verktyget skall kunna integreras med Maven

QuDo63 Verktyget skall kunna köras i Docker

QuDo64 Verktyget skall kunna integreras med Github

QuDo65 Verktyget skall kunna köras på Jenkins

QuPr66 Verktyget skall kunna visualizera den uppmätta exekveringstiden enligt bild E.1

QuDo67 Verktyget skall kunna köras på TestNG tester

QuPr68 Verktyget skall kunna visa om en query är lazy

QuPr69 Verktyget skall kunna visa en queries select statement

QuPr70 Verktyget skall kunna visa om en query är långsam på grund av indexering

FuPr71 Verktyget skall länka tillbaka till rätt pull request

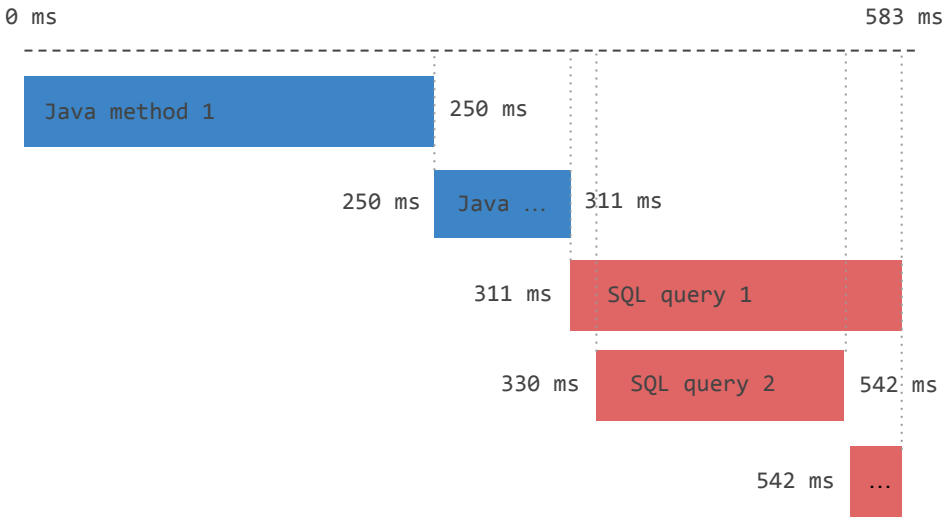

Figure E.1: The figure shows a method of visualizing the stack trace.

Appendix F
Core Requirements

F.1 Core Requirements

QuGo1 Verktyget skall minska felsökningstiden med ___%.

QuDo12 Verktyget skall vara automatiserat.

FuDo14 Verktyget skall kunna identifiera långsamma queries i testning.

QuGo17 Verktyget skall kunna ge feedback från byggstadiet inom ___ minuter.

QuPr25 Verktyget skall kunna köras lokalt.

QuGo27 Verktyget får inte belasta systemet ytterligare.

F.2 Most Valuable Requirements

FuDo3 Man skall kunna välja vilka slow queries som skall monitoreras.

FuDo4 Man skall kunna specificera vad en slow query är.

QuDo5 Verktyget skall kunna ge feedback från monitorering inom 24 timmar.

QuDo6 Verktyget skall kunna ge feedback från testning inom 30 minuter om det körs på Jenkins.

FuPr9 Verktyget skall kunna ge information om en slow query i GitHub.

QuDo13 Verktyget skall vara ett byggsteg i Jenkins.

FuDo15 Verktyget skall kunna varna om långsamma queries i testning.

FuPr19 Verktyget skall kunna identifiera vilket steg i tech-stacken som är långsamt.

FuPr21 Verktyget skall kunna notifiera om en specifik plats i koden som är långsam.

QuPr24 Verktyget skall kunna användas för enskilda commits.

FuDo26 Verktyget skall kunna göra och visa en check i GitHub pull request.

FuPr28 Man skall kunna ställa in gränser för vad som räknas som långsamt.

FuPr30 Verktyget skall kunna skilja mellan långsamma queries och många jämlöpande queries.

QuDo38 Verktyget skall kunna köras både i test-stadiet och i produktion.

QuDo41 Verktyget skall kunna konfigureras i kod.

FuPr43 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen i Jenkins.

FuPr44 Verktyget skall kunna testa endast de ändringar som har gjorts i kodbasen lokalt.

QuPr48 Verktyget ska inte förlänga testtiden med mer än 10% när det körs lokalt.

QuPr50 Verktyget skall filtrera bort onödigt brus.

QuDo54 Performance tester ska kunna skrivas med ett standardramverk, t.ex. TestNG eller JUnit.

FuPr55 Verktyget skall kunna mäta tider i callstacken.

QuDo59 Verktyget skall kunna köras på MacOS.

QuDo60 Verktyget skall kunna köras på Linux.

QuDo61 Verktyget skall kunna användas på Javakod.

QuDo62 Verktyget skall kunna integreras med Maven.

QuDo64 Verktyget skall kunna integreras med Github.

QuDo65 Verktyget skall kunna köras på Jenkins.

QuPr66 Verktyget skall kunna visualizera den uppmätta exekveringstiden enligt bild F.1.

QuDo67 Verktyget skall kunna köras på TestNG tester.

FuPr71 Verktyget skall länka tillbaka till rätt pull request.

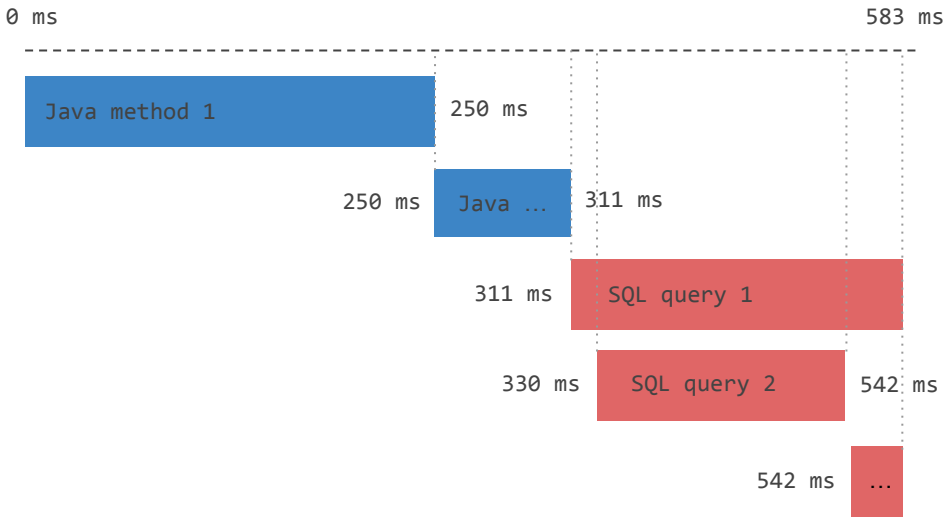

Figure F.1: The figure shows a method of visualizing a stack trace.

F.3 Nice to Have Requirements

QuGo2 Verktyget skall minska tiden det tar att åtgärda problem i produktion med ___%.

FuDo7 Man skall kunna välja vilka tabeller som skall monitoreras.

FuDo8 Verktyget skall kunna visa vilken del av databasen en slow query påverkar enligt krav xx.

FuPr10 Verktyget skall kunna ge information om en slow query i Jira.

FuPr11 Verktyget skall kunna ge information om en slow query i Jenkins.

FuPr16 Verktyget skall skicka en notis ifall en query är problematisk.

FuPr18 Man skall kunna definera tabeller i en databas som inte får låsas.

FuPr20 Verktyget skall kunna ge information om vilka tabeller och fält(kolumner?) som orsakar en slow query.

FuPr22 Verktyget skall kunna identifiera lösningar till långsamma queries.

QuPr23 Verktyget skall inte visa hela sidor med information.

FuPr29 Verktyget skall kunna stresstesta koden.

FuPr31 Verktyget skall kunna load-testa.

FuPr32 Verktyget skall kunna summera totala tiden för återkommande queries.

FuPr33 Verktyget skall kunna ge standardavvikelse för tiden en query tar.

FuPr34 Verktyget skall kunna ge den snabbaste tiden en query tar.

FuPr35 Verktyget skall kunna ge den långsammaste tiden en query tar.

FuPr36 Verktyget skall kunna ge queries/minute under en viss tidsperiod.

FuPr37 Verktyget skall kunna generera Jira issues om en slow query automatiskt.

QuPr39 Verktyget skall ta ca. 1 minut att använda.

QuPr40 Verktyget skall kunna ge feedback inom ca.1 minut om det körs lokalt.

FuPr42 Automatiskt genererade Jiror skall tilldelas personen som orsakade slow queryn.

FuPr45 Användaren skall kunna filtrera bort för stunden orelevanta queries.

FuPr46 Man ska kunna välja bort de versioner som är testade och inte innehåller på långsamma queries.

FuPr47 Verktyget skall ha möjlighet att ignorera en viss version av query genom opt-out.

FuPr49 Verktyget skall kunna skilja användaqueries mot interna queries.

QuPr51 Verktyget skall endast skapa en Jira per performance issue.

QuDo52 Verktyget skall kunna integreras med Jira.

FuDo53 Verktyget skall kunna monitorera produktionsmiljön.

QuDo56 Testningen skall vara tillräckligt snabb för att kunna användas i Test Driven Development.

FuDo57 Verktyget skall automatiskt kunna identifiera vilka tester som ska köras.

FuDo58 Utvecklare skall kunna specifiera vilka tester som ska köras.

QuDo63 Verktyget skall kunna köras i Docker.

QuPr68 Verktyget skall kunna visa om en query är lazy.

QuPr69 Verktyget skall kunna visa en queries select statement.

QuPr70 Verktyget skall kunna visa om en query är långsam på grund av indexering.