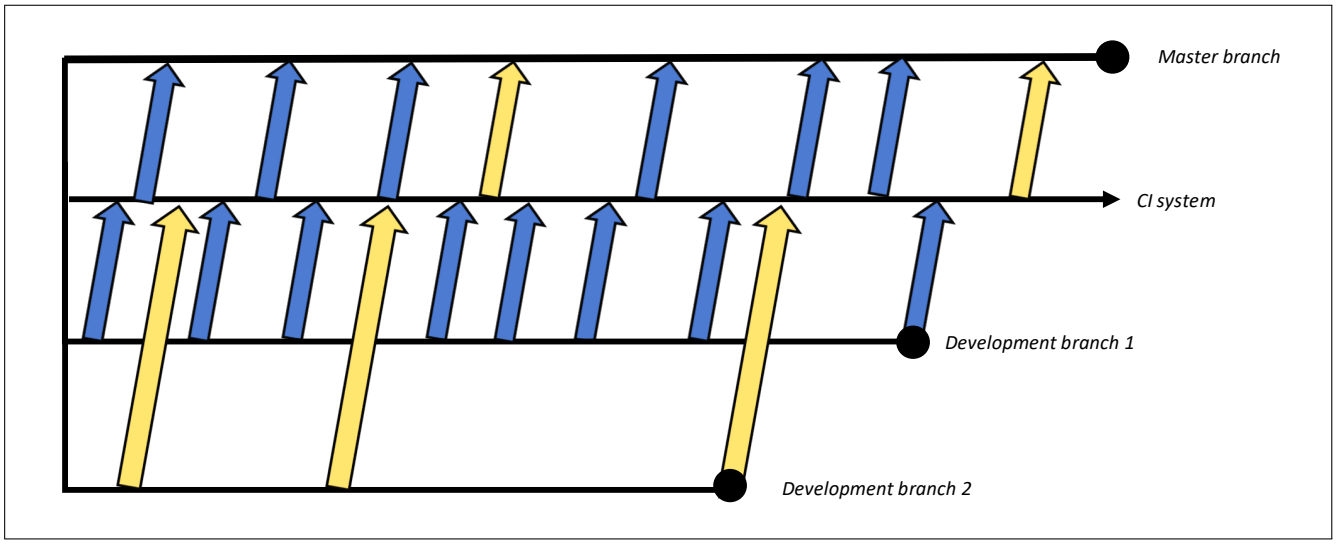
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



An Investigation of Integration Debt in Continuous Integration

A design science study

Master's thesis in Software Engineering

Pooriya Balavi: *gusbalpo@student.gu.se*

Kevin Rasku: *rasku@student.chalmers.se*

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

An Investigation of Integration Debt in Continuous Integration

A design science study that explores the notion of integration debt and its causing factors

KEVIN RASKU

POORIYA BALAVI

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

An Investigation of integration debt in continuous integration

Supervisor: Jennifer Horkoff, Software Engineering division at Chalmers university
Supervisor: Terese Besker, Software Engineering division at Chalmers university
Examiner: Robert Feldt, Software Engineering division at Chalmers university

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Acknowledgements

We would like to thank the involved supervisors, Jennifer Horkoff and Terese Besker, that guided the researchers in the right direction. We also like to thank one of the developers from the involved company, Andreas Lindhé, who acted as an advisor throughout this study.

Pooriya Balavi, Kevin Rasku, Gothenburg, June 2019

Contents

List of Figures

List of Tables

Abstract

Continuous integration is a commonly used method in software development. It is used to improve the development process by prescribing frequent code integration and testing and thereby creating a mechanism for early feedback. To optimize the use of continuous integration in large scale development in regards to code integration, this study introduced a new term called integration debt. The concept is borrowed from the notion of technical debt, applied to continuous integration and the factors that are involved in this integration debt (which happens due to unintegrated code) are investigated. In this study, a set of integration debt factors is defined and the relationships between the factors are mapped and analyzed. This is done through design science research methodology where the artifact is the mapping of integration debt factors and the set of factors itself. Using various data gathering methods, a set of factors that are involved in integration debt are identified and validated. These factors were lines of code in commits, the number of modified files in commits, code churn, frequency of commits, code dependency, development overlap and lastly the factors related to the organization's CI system.
Additionally, this study investigates, and evaluates, a potential threshold value for one of the factors where the threshold is for when that factor reaches a significant risk to integration. Defining a general and accurate threshold proved to be difficult which suggests that integration debt factor thresholds require significant effort to define. This could perhaps be investigated further in future research. Consequently, the findings of this research study build a foundation for integration debt and how it could be mapped and measured. This can aid companies that work with large scale development, to better manage the integration debt level across their project by taking appropriate actions to avoid large accumulated debt levels. This potentially lowers the costs for companies and reduces the extra effort required to resolve integration problems during development.

Keywords: *Continuous integration, Integration debt, Repository mining, Software metrics, Code integration, Merge conflict, Automated test and build*

1
Introduction

In the field of software engineering, there is a well-established notion of technical debt [1]. It refers to the debt that a developer takes on when implementing a suboptimal solution which may save time and costs in the short-term but is wasteful in the long-term [2]. This wastefulness and the potentially increasing difficulty of correcting it, of the suboptimal solution over time can be viewed as the interest that is paid on the debt. In this research study, the concept of technical debt is used and applied to continuous integration (CI) in large scale software development to optimize its efficiency. Continuous integration practice is a widely adopted approach of routinely integrating code change into a shared repository using a dedicated system. CI suggests testing of the change as early as possible before its merge to the mainline. It also proposes frequent code integration from developers, ideally at least once a day [3].

The referred debt is a particular form of liability, named *integration debt* which has not yet been thoroughly investigated. The term encompasses any code that exists anywhere in a project, which is not already integrated into the mainline (master branch). In other words, when software developers stay behind the mainline by not integrating local code to mainline and fetching the latest code from it. Mitigating the accumulated debt is done by integrating (unintegrated) code to the mainline, and a higher level of debt will, per our definition, have a higher risk of causing integration problems such as merge conflicts. To minimize integration problems, it is in the best interests of the developers to know how much integration debt they can take on and what level of integration debt they currently have as well as knowing which activities can add up to integration debt. It is also worthwhile for project managers to get insights on where the debt is being accumulated in their project repositories. By knowing that, project managers can take appropriate actions before these debt levels are too large to handle. Piled up debt across different development branches can be costly in terms of affecting the duration of test/build and causing integration problems which require additional time and effort.

Research initiatives of technical debt and continuous integration and their effects on software development have received considerable attention in previous research as will be discussed in the next chapter. However, the idea of what integration debt is and how it can be measured within the context of continuous integration development has not yet been investigated. Thus, there needs to be a way to measure integration debt across a project and find measures or metrics to allow visualization

of the current debt level. This means there needs to be a threshold level or recommended value (or range of values). This is what our study aims to achieve, a set of metrics for integration debt and a recommended value or range of values for this set of metrics. Moreover, to be able to measure this accumulating debt, the factors that affect integration debt need to be investigated and identified based on their level of significance.

The company that takes part in this study, allows the researchers to examine one of its large scale project repositories and get insights from its developers, in order to get a better understanding of integration debt and its causing factors. The company has reportedly shown concerns regarding the issues with the code that is being developed in isolation for too long. This is mainly because when the master branch is not updated frequently, other developers will not have access to the latest features being developed and that leads to various problems such as; integration complications, additional effort to resolve merge conflicts and production delay. Thus it is interesting to measure and monitor the accumulating debt to be able to minimize costs and extra effort during development.

In this study, there were several methods used to analyze integration debt factors, including review of existing literature, interviewing developers from different teams, data mining of the repository and carrying out a questionnaire to validate the final affecting factors. The results from this study will be useful to any developer working in a common code base who want to optimize their use of continuous integration. This could potentially be most valuable in larger projects where a large number of developers regularly integrate their code and where there are potential integration problems that may be very costly. It is important to mention that the discovered factors can likely be generalized (to some extent) to other large scale projects which utilize continuous integration, however, the threshold (recommended debt value) is likely specific to the organization that is taking part in this study.

This research is structured as follows: there are eight chapters where each has its defined content. Chapter 2 discusses the involved concepts related to this study (i.e. CI, technical debt and software metrics) and digs deeper into how they are related to integration debt. It also goes through the related work and interprets how their findings are applicable in this research. Chapter 3 and 4 explicitly define what this research is after and what methods are used to get to the conclusion. In chapter 5, the collected information regarding integration debt is presented. These results are from literature, interviews, repository mining, and questionnaire. Chapter 6 aims to use the data from the results section to discuss and interpret the collected data. This chapter reflects on the findings and in what ways they are significant. Additionally, threats to validity and difficulties encountered throughout the study are discussed. Chapter 7 gives a summary of the concluding results and mentions how this research could be further continued.

2

Background

Since there have been no in-depth studies on the topic of integration debt, the notions of continuous integration, technical debt, and software metrics become the foundation of this research. Therefore they needed to be thoroughly explored. In this section, these concepts related to integration debt are investigated, including how they affect this integration debt.

2.1 Continuous integration

Continuous integration has become one of the most widely adopted practices of software development [4]. The word *continuous* refers to the absence of time-constraints and *integration* emphasizes the aggregation of software parts in one central storage [5]. The concept of continuous integration was originally suggested by Grady Booch in the context of object-oriented design, where he proposed: *"At regular intervals, the process of continuous integration yields executable releases that grow in functionality at every release"* [6]. A decade later Martin Fowler took the idea further by stating that *"each integration should be verified by an automated test and build to detect integration errors as quickly as possible [7]."* The core definition of CI forms the foundation of our study since integration debt should be confined to the scope of CI, and in our case, we limited the scope to technical factors that are measurable within the CI-system (which also includes version control). Debbiche et al.[8] performed a case study in a very similar company to the one in our study where they looked at the challenges involved in adopting continuous integration. They found that things like code review, code dependency and the testing stage (in CI) all affect the implementation of continuous integration. These are interesting for us to consider as potential integration debt factors since they are within the scope of this study and they either are integration problems themselves or they cause integration problems.

This practice is reported to enhance release frequency, predictability and mitigate integration risks [3] as well as boosting developers' level of productivity [9] among many other benefits. CI offers defined principles of development but its application

varies in the industry depending on many factors, such as the size of the project, number of developers, culture of the company, the longevity of the project and other factors. This has led software developing organizations to tailor their CI-flow in a way to benefit them best by using the right tools and technology. Thus tooling that facilitates CI can vary across organizations but the practice itself does not particularly need a tool [7].

CI encourages developers to merge their code frequently to the main branch of a shared repository and test the newly added code before merging. The frequency of commits may be affected by the difficulty of the task at hand, the CI work-flow of an organization and the productivity of developers. However, the suggested number of commit intervals is at least once per day [7, 8] in order to minimize merge conflicts but often in reality that is not the case where the idea of frequent integration is neglected by developers [3].

Another major CI principle emphasizes continuous test & build of the newly merged code [7]. In 2014, Ståhl and Bosch [3] examined the details of how an organization can successfully adopt the practice of continuous integration. One of the most important reflections was the importance of testing and building the newly merged code in a large scale software development, where cross-functional teams work closely together to develop features. Regarding the duration of test & build, they mention that *"a too long duration means continuous integration could start to break down. This duration must be quick enough to allow the CI server to keep up with the changes and return feedback to software engineers while their memory of changes is still fresh. One possible solution is to separate quick tests from slower ones and provide incremental feedback."* The significance of automating this procedure is also mentioned where it is said to increase the efficiency of the merging process on a larger scale and aid to improve the feedback time, maintaining a healthy repository and ultimately eliminate waste. In this research study, the main focus is on investigating integration debt in a large scale software project where CI practices are applied. In chapter 5.2, the CI-system of the company that is involved in this study is broken down and later on is investigated to see how the integration mechanism could affect integration debt as well as whether any aspects of the CI flow directly or indirectly affect integration debt levels.

2.2 Technical Debt

In 1992, Ward Cunningham first discussed the notion of technical debt (TD) in software development [2]. He referred to it as *"a trade-off between writing clean code at the cost of more expenses and potential delayed delivery"*. In other words, it can be referred to as the extra effort and cost that needs to be done to compensate for the *"quick and dirty"* development that was done to benefit the short-term goals [10]. By choosing a suboptimal solution, a developer takes on interest that has to eventually be paid. This interest can be paid down by refactoring, choosing better

design decisions or otherwise fixing the raised issues but if ignored, the developer will continue paying the interest and allow the debt to be accumulated. In extreme cases where the debt has reached uncontrollable levels, *bankruptcy* could happen where the code has become unsustainable and has to be thrown away or rewritten completely [11]. Technical debt has been estimated to waste a significant amount of development time [12]. In most cases, the debt does not immediately show an effect on the software and a fast time-to-market development approach may even give the illusion of early return of investment for a company. Though in reality, the accumulated debt usually presents itself on later stages, like in the release cycle or integration where it can impede further development.

There have been numerous research studies around the topic of TD, including what it consists of, its drawbacks and how it could be controlled. Though in the context of this research, existing studies about measuring the development process and the debt related to it appear to be the most related. Tom et al. [11] talk about different dimensions of TD, including *environmental debt* where they mention *"Technical debt can also manifest in the environment of an application, which includes development-related processes..."*. Thus integration debt could be viewed as an environmental debt within the context of technical debt. Since continuous integration is a development process and is not directly concerned with writing the program code, this type of environmental debt is arguably the closest currently explored topic to integration debt within the domain of technical debt. Alves et al. [13] specify *build debt*, *test debt*, *process debt* and *infrastructure debt*. These could all be viewed as being related to integration debt since continuous integration involves building and testing.

Technical debt is inevitable and it is unrealistic to set a goal of having absolutely zero technical debt [1], but if the notion and its impacts are well understood, communicated and appropriately managed, it can make a significant impact on success of a developing software in both a short-term and a longer-term perspective [14][10]. Avoiding technical debt may increase time-to-market but it allows low debt interest payments. Therefore companies are continuously optimizing their development approach to systematically lower the level of technical debt in their projects. For this study, the concept of technical debt is used and applied in the context of continuous integration to optimize the use of CI and be able to monitor the accumulating integration debt. Measuring and monitoring technical debt has proven to be beneficial for organizations [15], therefore the concept of TD is valuable to apply to CI in order to evaluate integration debt levels throughout a project.

2.3 Software Metrics

Many different metrics are used to measure different aspects of software and software development. These range from more abstract measurements of process and quality to objective measurements of intrinsic code attributes like complexity and coupling [16]. Since this study aims to measure the factors involved in integration debt, it is

suitable to reuse existing metrics that can measure different aspects of software and software systems. Any existing software metrics related to the different factors are interesting as long as they are not outside of the scope of our study.

The scope of the factors being considered is that factors that are measurable within the CI-flow fall within the scope, whereas factors that are external, for instance; management factors are outside of the scope. Bearing this limited scope in mind, the types of software metrics that are relevant here are ones related to the continuous integration system, the version control system and the code that is being committed. This would include any potential metrics for things like frequency of commits, code churn, branching, build time, code size and code complexity.

While some of the existing software metrics from literature are of interest and can cover some of the integration debt factors, there is still a need to create or redefine certain metrics to cover certain factors. In one of the existing studies about a type of software metric (dependency), Staron et al. [17] goes more in-depth on software dependency and mention: *"dependency showed that in practice the design of those two dependent components needed to be synchronized otherwise a risk of integration defects (hard and costly to find) can be significant"*. A risk of integration defects is close to the definition of integration debt which makes the study interesting for this research. Their study introduces a method for measuring dependencies within a code repository which could be useful for measuring dependency as an integration debt factor.

Code churn is another type of software metric that is suspected to affect integration debt. In a study about the need to measure code change/churn, Hall and Munson [18] discuss that "*We would like to be able to measure the rate of change of program modules so that we might establish some notion of the rate at which faults might be injected into the system via a change in complexity.*" They also mention ways to measure it. Nagappan and Ball [19] took the topic further by finding that relative code churn affects defect density. These papers are relevant for our study because they suggest that code change or churn, as well as LOC and files changed, are factors we should consider in integration debt because it affects the rate at which faults might be injected. Faults may lead to failed tests or merge conflicts which in turn leads to delays in the integration, thereby adding up to integration debt. Perry et al. [20] studied software engineers developing code in parallel. They found that a higher degree of parallelism leads to more problems and defects. Especially if the changes are interfering (they overlap) and if they are in close temporal proximity to each other. These findings are important for us to consider since it shows that code overlap in commits, frequency of commits and merge conflicts are potential factors in integration debt.

In another study about the effects of branching strategies on software quality, Shihab et al. [21] use branch activity, branch depth, and branch scatter as factors related to branching. They found that activity and scatter have a significant effect on software quality. This study shows that factors related to branching could be interesting to

investigate as integration debt factors since it has an effect on software quality which in turn may affect integration. Lastly, in a study where the term integration debt is used, Rogers argues that: *"By delaying the full system integration, teams are potentially building up an integration debt that will be painful to resolve when the integration finally happens."* [22] This is more or less what integration debt is and why it could be useful to measure. Rogers, however, does not go further into integration debt and what causes it as well as how it could be measured. That is where our study comes in to further explore the notion of integration debt, the factors involved it and suggest a potential solution when it comes to modeling (or mapping) it. Additionally in Rogers' paper, reducing dependency through modularization is mentioned in relation to integration debt. This informs us that we should consider investigating dependency as an integration debt factor.

As a result of the review of the existing literature on software metrics, it became clear that there exists no integration debt metric that is explicitly defined or investigated. From the existing software metrics, there are none for debt specifically concerning continuous integration. Some aspects of integration debt are covered by other metrics but there exists no holistic understanding of it. This is the gap in the current body of research which our study aims to address.

3

Research Questions

This study aims to appropriately define the term integration debt by exploring the topic and more importantly identify the factors involved in it. This requires investigating the software development metrics and the CI system's factors that have an impact on integration debt. The factors involved in integration debt are things that cause code to stay unintegrated for a longer time, and things that cause or increase the risk of integration problems. Furthermore, this study aims to investigate a potential threshold value for when the accumulated integration debt for a factor reaches a problematic level and when perhaps appropriate action may be required to mitigate the debt level.

There are three research questions (RQs) that this study intends to answer. These questions were designed in a way to enable the researchers to tackle the company's problems within their CI-system. The aim was to identify what integration debt consists of and how it could be mapped or measured such that the company can apply it to their CI development in order to improve it. The third question builds on the findings from the second question. Similarly, the second question is a continuation of the first RQ. The following is the first research question for this study:

RQ1: What are the factors that impact integration debt?

This first research question forms the foundation of this study as it aims to provide an understanding of what integration debt is and what it consists of. This requires identifying the factors that can affect integration debt within our scope.

RQ2: How can integration debt be refined, or broken down, in order to be measured?

The second research question builds on the first research question by taking the factors involved in integration debt and assessing how they could be refined or measured. The answer will be a mapping of factors for integration debt, which includes the factors from RQ1.

RQ3: Can thresholds for integration debt factors be determined?

The purpose of the third research question is to determine if thresholds for the integration debt factors can be created. A factor may have a threshold for when the value becomes a potential problem and this may be useful to determine, which is why this research question is of value. The thresholds may vary depending on the organization using continuous integration. In this study, only one of the factors (LOC) is investigated in terms of its potential threshold value. It is examined to determine if an accurate threshold can be found, meaning if the value for when the factor becomes a significant problem can be predicted.

4

Research Method

This study aims to investigate what integration debt consists of and how it can be refined and broken down (and mapped), and the type of research methodology chosen is design science. The artifact being designed is the mapping of integration debt factors and the set of factors itself, along with their associated metrics. The artifact also consists of a potential threshold value that states the dangerous level of debt in regards to an integration debt factor. To develop and evaluate the artifact, the study is conducted together with a company in the industry. This provides the opportunity to investigate integration debt factors in a real-world continuous integration setting and also allows for feedback to be gathered from the developers at the company.

4.1 The company involved in the study

This study is done in collaboration with a software company that works with large scale software development. Upon request, the company's name is not revealed in this study and some sensitive information has been redacted. The company is active in several industries. They develop software using common software engineering methods such as agile and continuous integration. There are several cross-functional teams at the company that collaborate to develop code which is merged into common repositories.

The company's developers, code repositories, CI-system, and its historical data are utilized for data gathering. More specifically, the CI-system provides data about the CI process, the repositories are mined for additional data, the developers are surveyed both to gain more information about the CI process and integration debt factors, as well as to evaluate the findings. The large-scale software development and the availability of resources (such as repositories and developers) made this company a good fit for this study. In addition, there was also an established CI-system and an active interest from the company to learn about integration debt which further increased the suitability of the company for this study.

4.2 Design Science

The methodology of design science involves designing and studying an artifact within a context [23]. The artifact is meant to improve something within that context. The method is iterative and three of the main tasks involved are problem investigation, treatment design, and treatment validation.

For this study, the context is software development using continuous integration and the artifact is a mapping and its set of factors and metrics for measuring integration debt (Mapping simply refers to diagram that displays how integration debt factors could affect the accumulating debt). The problem in this research involves investigating which factors are involved in integration debt, how they can be measured and what a reasonable mapping or framework for them could be. The treatment design is the design of the integration debt mapping/factors/metrics and the treatment validation is the evaluation of these.

The study is divided into three iterations where each includes investigation, design, and evaluation. Data gathering and data analysis are part of each iteration using different methods. The duration of each iteration was approximately five weeks and the information gained in one iteration, or one stage of an iteration, was built upon in the next one. A complete description of the iterations (in chronological order) and the data gathering/analysis methods employed follows in the sections below. An overview of the iterations and their activities can be seen in figure 4.1.

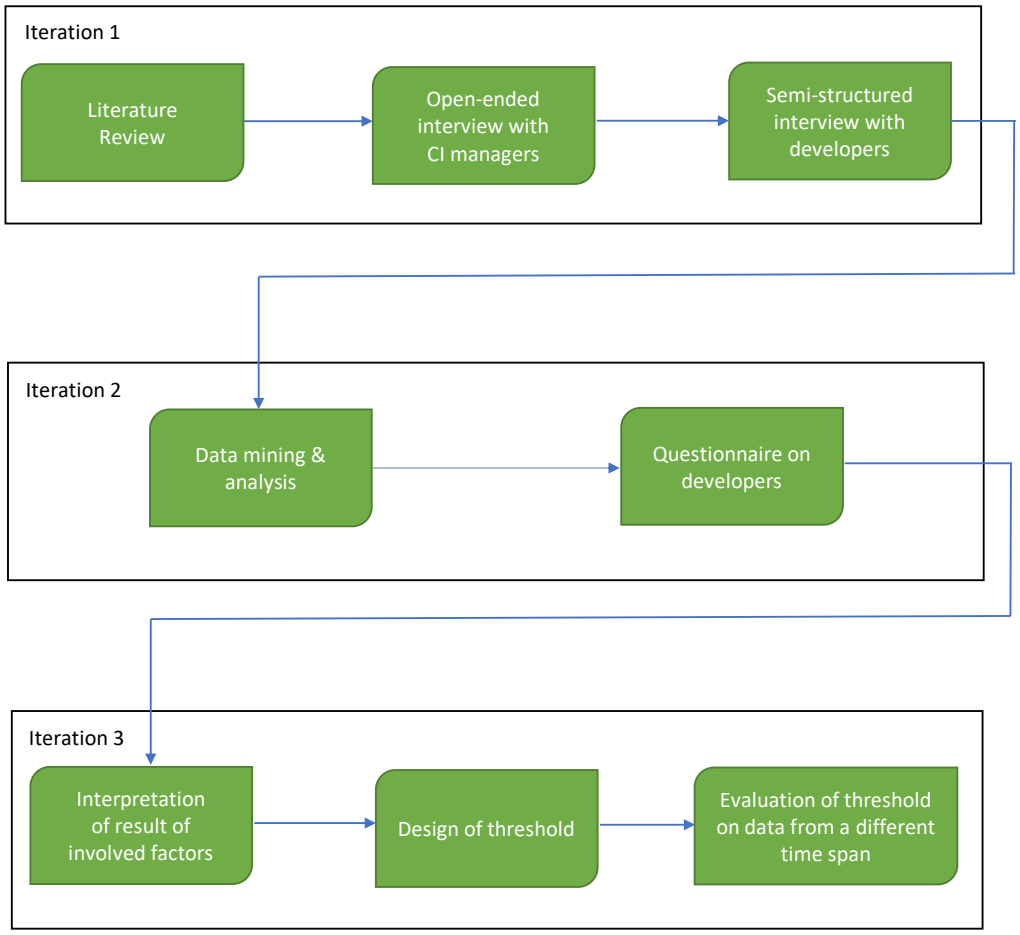


Figure 4.1: Figure of the iterations & activities performed in this study

4.2.1 Iteration 1

Literature review

A review of relevant existing literature was performed in order to investigate the problem domain and also to know what the current state of research is on the topic of integration debt. It provided an overview of what had already been done, and thus what is not needed to be done again as well as what could potentially be interesting to do. This helped us with defining our research questions and the scope.

The literature review was also used to find ideas and create assumptions regarding potential integration debt factors. The literature on continuous integration, technical debt, and software metrics provided us with a number of potential integration debt factors that have been proven to affect CI or parts of CI. For instance, the code metric of dependency affects the number of test cases needed for thorough testing, which in turn may affect the testing that is part of the CI-process, that might make

it a factor involved in integration debt. The factors that were generated through the literature review were investigated further through the other methods, such as interviewing, questionnaire and repository mining.

The search for relevant literature for the review was done primarily through digital publication libraries such as IEEE Xplore [24], ACM [25], and SpringerLink [26]. Google Scholar [27] was also used to search for relevant studies. Examples of search terms used are continuous integration, integration debt, technical debt, and software metrics. These search terms were also combined, both with each other and with the integration debt factors that were being investigated, for instance, lines of code and frequency of commits.

Open-ended interview with CI managers

An open-ended interview with CI-managers (at the company) was held to get a preliminary understanding of the CI-system and more specifically their step-by-step procedure of merging code, as well as current issues of their CI-system. This took place in the first iteration and the purpose was to get an overview of the CI-flow and be able to later use this information together with other findings in order to judge whether any stages in the CI-system affect integration debt. The CI-managers also gave their views on potential integration debt factors within the CI-system.

Interviews

Interviews are conducted to get feedback about the factors and potentially find new ones. There were ten interviews with the developers at the company that were used to gather data about their CI-process, problems with it and what they think could be potential factors involved in integration debt. The developers were chosen through convenience sampling due to availability reasons. Each interview took around 30-45 minutes, which were formed to be semi-structured and the main aim was to get ideas about potential integration debt factors, as well as asking for the developers' views on the factors we came up with from the literature review. The interview questions can be read in appendix A.1 and further information regarding the results of the interviews is shown in the next chapter.

For the interview answers, they were individually transcribed and coded using thematic coding as described by Dyba & Cruzes [28] and Saldana [29]. The transcriptions were read through and for each type of topic discussed, a theme was created. Each theme has a set of codes where each code came from patterns in the interview answers. The codes have specific interview responses associated with them. This information is presented in section 3.5 of the report. Organizing the interview data in this way allowed us to gain a complete view of the responses and also enabled us to quantify the different responses, for example how many thought that a specific factor affects integration debt. The coding was done by the two researchers in isolation and then cross-checked against each other to reduce bias. Every factor that was found to be relevant to the concept of integration debt was noted, and a list of

integration debt factors was compiled.

4.2.2 Iteration 2

Data mining & analysis

Data mining allowed for specific data on anything relating to commits, time of code merges, occurrences of merge conflicts, and information regarding active branches to be gathered. The important information regarding the company's CI-flow was stored by the company and the data was available to researchers of this study. This information was concerning commits that were being submitted to be merged to the master branch and consisted of lines of code of commits, the number of modified files in commits, time stamp of commits, owner of commits, etc. Conducting data analysis on CI data, provided the opportunity to investigate correlations and relationships between the factors. The data was analyzed to see which factors affect integration, how much they affect it, and how the different factors are connected. Data mining also allowed for some validation of the existing data and whether the previous findings are accurate. Potential new factors can also be found in this stage. Although it is important to mention that often the existing historical data from the CI flow did not have all the necessary information and additional repository mining was required. For example, if the Gerrit [30] change-IDs of commits that caused merge conflicts could be retrieved from the CI-system but not the lines of code, then a small program [A.3] was written to fetch that information from the repository. Gerrit is an open-source code review tool that is built on top of the Git version control system [31], which will be further discussed in the Results chapter.

The approach to data mining was based on the results from the previous iteration. From that iteration (literature review, interviews with developers, and the interview with CI-managers), a set of assumptions regarding the potential integration debt factors had been established and a basic mapping of their relationships had been created which is shown in Figure 5.1. The idea was to further analyze and validate these findings through iterations 2 and 3. The factors that were found to add up to integration debt (and thereby increase the risk of integration problems) were investigated for correlations with the factors that were themselves part of the integration problems. For example, in iteration 1, the factor of lines of code was found to be an integration debt factor that could worsen or increase the risk of merge conflicts. As such, the correlation between lines of code and merge conflicts was investigated to check if, and how much, of a factor lines of code actually is.

Statistical analysis in the form of finding correlations was performed on the different factors using the CI/repository data as mentioned previously. It was necessary to do this in order to flesh out the set of factors from iteration 1 as well as to lay the groundwork for being able to refine and map integration debt, as well as to investigate integration debt factor thresholds.

The statistical analysis is conducted using R and RStudio [32]. The measures used for correlations are Pearson's correlation coefficient and Spearman's rank correlation coefficient [33]. These were chosen as they are commonly used and fairly simple methods of correlation analysis. This was deemed suitable as this is but one method of investigating the factors and their relationships in this study, and not the sole focus.

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

Figure 4.2: Pearson's coefficient formula

$$r_s = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

Figure 4.3: Spearman's rank coefficient formula with no tied ranks

Pearson's is parametric so if the data is normally distributed then it is used, if not then Spearman's is used instead. To test if the data is normally distributed, a normality test is used. Pearson's coefficient measures the linear relationship between two variables whereas Spearman's measures the monotonic relationship between two variables. Their usage also depends on whether a data set has outliers or not. When assessing if there are significant unremovable outliers, then Spearman's is preferred because Pearson's will be skewed more by the outliers. Spearman's and Pearson's are two statistical correlation measures that are appropriate to use in this study to measure the impact of the potential integration debt factors and assess whether they really have a significant effect on integration debt or not.

$$W = \frac{(\sum_i^n a_i x_i)^2}{\sum_i^n (x_i - x_m)^2}$$

Figure 4.4: Shapiro-Wilk normality test

The results from the analysis will show the correlations between the variables investigated. To interpret the correlation values (e.g. what is considered a strong or weak correlation), a suggested guideline is used [34] for evaluating the outcome of Spearman and Pearson values. The outcome of these two measures is between -1 to +1 indicating a positive or negative correlation between two variables (0 means no correlation). Here is the rather general interpretation of the coefficient value suggested by Evans in 1996 [35]:

- Very Weak: 0.00 - 0.19

- Weak: 0.20 - 0.39
- Moderate: 0.40 - 0.59
- Strong: 0.60 - 0.79
- Very Strong: 0.80 - 1.00

Questionnaire

A questionnaire was sent out to a large number of developers and CI-managers to validate the set of factors gathered. The number of participants was thirty and the number of questions in the questionnaire was nineteen. The template of the questions can be found in appendix A.5. Four of these questions were open-ended and the rest used a five-point Likert scale ranging from strongly disagree to strongly agree.

4.2.3 Iteration 3

Final integration debt mapping and threshold

At this stage, RQ1 and RQ2 have been answered and the set of integration debt factors have been designed along with the mapping of them. The next step is to answer RQ3. This is done by looking at the data gathered and extracting a threshold value for one factor. The threshold will then be evaluated using a new data set. The threshold was defined by identifying a level (of LOC) in the data where there is a significant increase in the proportion of merge conflicts and setting that to be the threshold.

Testing the threshold on separate data set

To evaluate the usefulness of the threshold, it is tested on a separate data set that is from the same repository but a different time-span. The threshold is evaluated on data from the latest two months of development. The purpose is to see if having the integration debt threshold would allow for any useful predictions to be made, such as if an increase in the risk of integration problems can be predicted.

5

Results

In this section, we present the results of our design science study. The results connect the findings to the research questions. To begin with, we have the results from the investigation into what factors are involved in integration debt (RQ1). Then there is the result of the mapping of the integration debt factors (RQ2). Finally, the results from the investigation into a potential threshold (RQ3) are presented.

5.1 Factors originating from literature

To put together an initial set of potential factors involved in integration debt, relevant literature was reviewed and the potential factors were noted. The result from this is shown as a set of initial potential factors displayed in Table 5.1, together with their sources and explanations.

Factors originated from literature		
Factor	**Source**	**Explanation**
Lines of Code	[19]	LOC is mentioned as a factor in causing defects which may affect test/build duration and merge conflict risk - and therefore also integration debt
Number of Files	[19]	Number of files is mentioned as a factor in causing defects, same as LOC
Freq of Commits	[20]	Temporal proximity of commits causes more problems and defects
Code Churn	[18], [19]	Churn is mentioned as the rate at which defects might be introduced
Dependency	[17], [22]	Dependency mentioned as factor increasing risk of integration defects, dependency also mentioned together with integration debt
Merge Conflicts	[20]	Merge conflicts mentioned as integration problem
Development Overlap	[20]	Overlapping development shown to cause integration problems
Branch Activity	[21]	Branch activity shown to increase defects
Branch Scatter	[21]	Branch scatter shown to increase defects
Branch Depth	[21]	Branch depth investigated as factor related to defects

Table 5.1: Overview of potential integration debt factors gathered from literature

5.2 Company's CI flow

In order to get more insights about the company's CI-system, an open-ended interview was organized with two of the company's CI-managers (i.e. DevOps and CI-administrators), who work with this machinery on a daily basis. The interview did not have a defined structure but was rather similar to an open-ended discussion aimed to collect information about the step-by-step flow of the CI-system. This information is later used to assist the researchers in determining whether any step of the CI-system affects integration debt. The findings show that the CI-system is designed to manage large amounts of incoming commits, carry out code quality control and detect faults, build each group of commits together with the existing code and provide the means of giving feedback to the code committers. However, due to confidentiality reasons, details of this CI-system cannot be displayed in this report. Thus only a general overview of it is shown in Figure 5.1.

The process mapping figure begins at the bottom where it shows how developers who work in diverse teams, commit their code from either a team branch, personal branch or a local computer where no version control branches are used. In the first phase of merging, a code is pushed to Gerrit code review[30], which is closely integrated with GIT [31], a distributed version control system. Gerrit also allows peer-review of changes before submission to automated test and build. If a commit is rejected at this stage, the committer is notified to fix the issue and push the code as new commit again. But if successful feedback from peer reviewers (who are usually committers' teammates) is revived, that commit can be submitted to Jenkins [36] which is an open source automation server. This is where automated test and build of the code begins. In the first stage of test and build, a commit is individually evaluated using many test cases. If one test fails, the commit is rejected to merge and its committer is notified through Gerrit where the feedback is sent to. Although if a commit passes all test cases, it goes to the next stage, where commits get grouped. The reason for grouping commits is due to the high number of incoming commits in the CI server and to best exploit the existing CI resources. After that, there is a second stage of test and build where additional test cases are done. If any of the commits in a group fails a test, the entire group is re-sorted in a new group and tested again. If a commit is re-sorted (i.e. goes in different groups) too many times, that commit is branded as the cause of group failure and the committer is notified through a feedback mechanism built on Gerrit.

If all the test and build stages are successfully passed, then the commit is successfully merged to the master branch. Excluding the Gerrit stage, the CI flow typically does not take a particularly long time. However, if there are problems during test and build, then the duration can increase. This could potentially add up to integration debt.

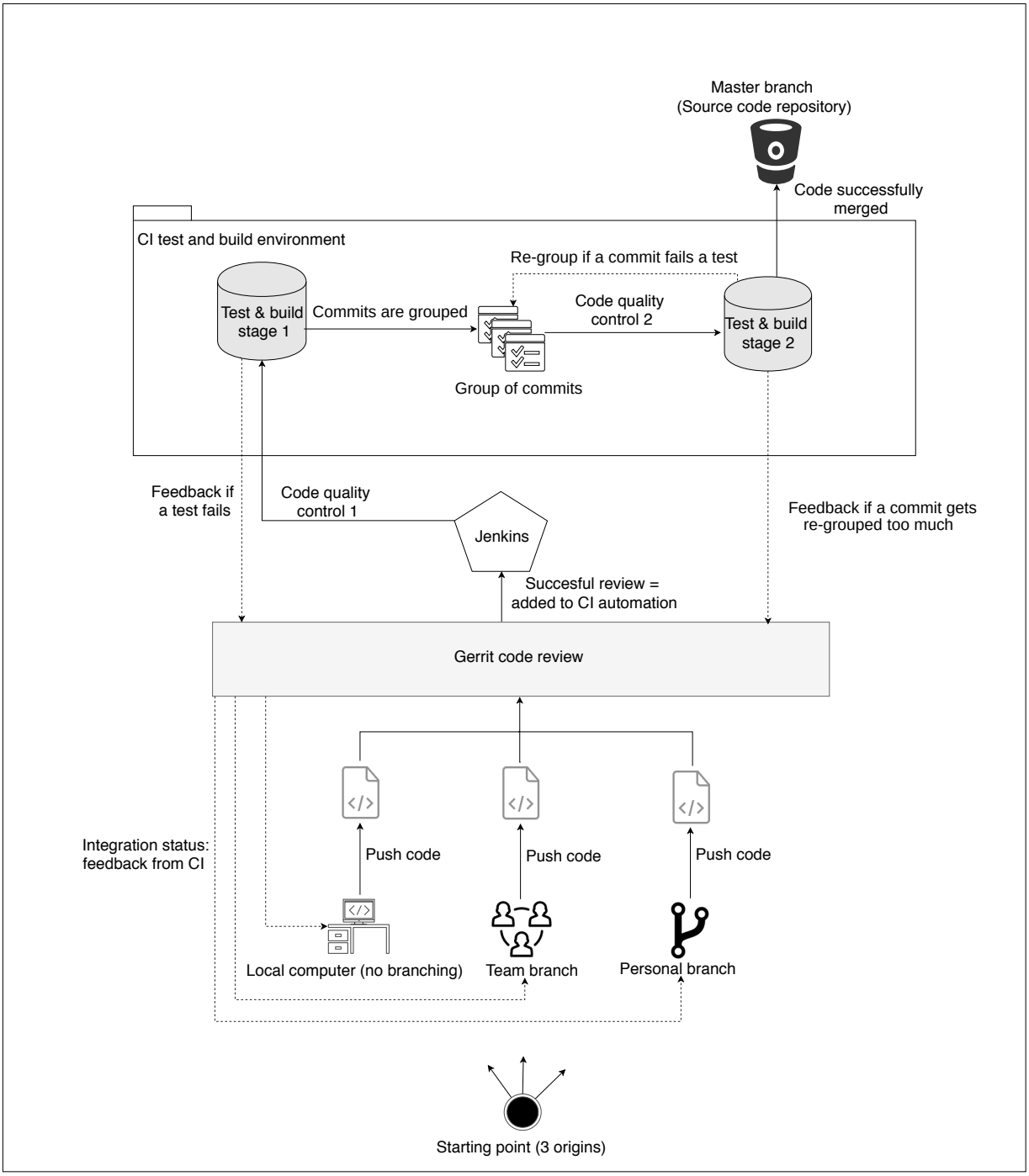


Figure 5.1: Process mapping of company's CI flow

5.2.1 Factors from open-ended interview with CI-managers

From the interview with CI managers at the company, there were a few factors that were brought up that seemed interesting to investigate as potential integration debt factors.

The first one was fault handling or debugging, meaning the time spent handling a failed test in the CI-system or a negative code review. This was brought up as a potential factor since it may affect how far away temporally someone is from integrating their code.

CI test and build cycle duration was also mentioned as a potential integration debt factor since it determines how much time it takes to integrate code to the mainline. There were several factors related to testing and building that were brought up separately but are included in, or relate to, the test and build factor. These include code dependency in commits, commits being queued in the CI-system and commits getting re-grouped (and retested). Production stops (for instance, if there were to be some problem that would halt the CI-machinery), and the amount of available CI resources (which may affect such things as how many simultaneous tests can be run) were also mentioned as potential integration debt factors.

5.3 Interviews with developers

Collecting qualitative information using interviews was necessary to be able to understand integration debt from the developers' point of view. Up until this point, all the existing knowledge is from the literature review and the open-ended interview with CI-managers. In the next step, the collected findings were evaluated through a set of 12 interview questions shown in the appendix (A.1). The interview questions were carefully designed to help the researchers get insights about the company's CI flow, its guidelines, its strengths and weaknesses, and most importantly to get some preliminary input about the causing factors of integration debt. Ten participants took part in the interview stage where each of the interviewees was asked the same questions. The structure of the interview was semi-structured, meaning that the researchers had questions prepared but some additional discussion was allowed with the participants when necessary.

Eight of the participants were software developers who worked with feature development, testing and code debugging. One of the participants was a scrum master and another was a CI-manager who maintained the CI-system in the company. It is worth mentioning that these participants were from eight different cross-functional teams where each worked on distinctive tasks. Every team has its own development process, for example; one uses Scrum while another could use Kanban.

5.3.1 Data categorization

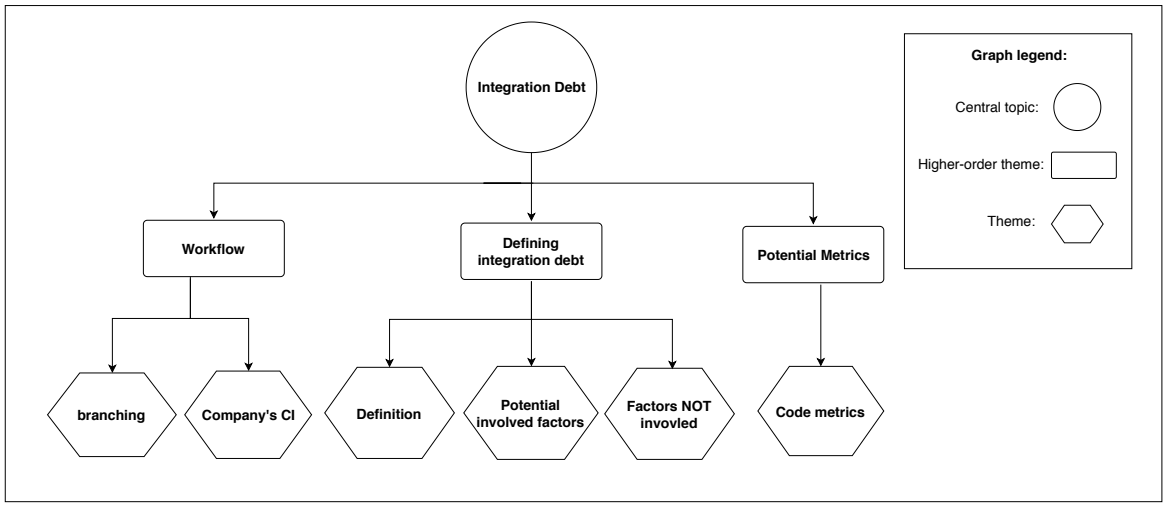


Figure 5.2: Categorization of the findings from the interviews

The Figure 5.2 illustrates the mapping of the results from the interview. After transcribing each interview as plain text, the findings were then categorized based on the model above. The data was divided into three main themes at the higher level presented in a rectangular shape. These higher-order themes are workflow, defining integration debt and potential metrics associated with integration debt. At the next level of the hierarchy, each major category is split into smaller themes displayed in a heptagon shape. Moreover, each theme is then divided into smaller codes which allows grouping the findings. These codes are mentioned in the appendix together with some of the findings (direct quotes from participants) that stand out the most in that code. The inconclusive answers or the ones that were too short to be considered are mentioned in the appendix.

The first category of interview results is workflow which aims to show information regarding the branching flow of the developers as well as the company's CI flow from participant's perspective. Each of these two themes is divided into codes and every corresponding answer from participants is noted. In the second main category that is about defining integration debt, the participants were asked about their opinion on the definition of integration debt and more importantly the factors involved in it as well as what does not affect integration debt. The last main interview category is dedicated to the potential ways of measuring integration debt from the participants' perspective.

5.3.2 Breakdown of the interview results

There were some quotes from the participants that stand out the most and contributed greatly to understanding integration debt and what does and does not affect it from the developers' point of view. An interesting finding from the interviews was that several participants were skeptical towards when CI-system is on pause, i.e. when production (merging process) is inactive due to technical reasons. They believed the delivery stop leads to different issues such as prolonging the integration process. As an example, one interview portrayed these issues as:

> *"When the wheel stops turning and there's a delivery stop and people can't get their commits through, the integration debt just keeps growing and growing and growing."*

When asked about code dependency in commits, 9 out of 10 participants thought that high coupling would increase integration problems and add up to extra effort and resources to resolve merge conflicts. One participant described the issues as;

> *"If you are waiting for someone's commits/feature to get merged, and your code is depended on another aspect that is under test. This could make you stay behind master branch. High coupling also could lead to merge conflicts."*

On the other hand, one participant's answer differed from the rest when he said:

> *"No, high coupling is not a factor. It is a technical debt but not necessarily an integration debt factor."*

The majority of participants (8 out of 10) believed that the fault handling process which is the process of identifying and removing errors from a piece of software (debugging) does not affect integration debt. They thought of debugging as part of their daily tasks. One interviewee described this as:

> *"If you get a negative review in Gerrit then you put the time in to fix it, same thing if your code doesn't pass then it's time to fix that. But I wouldn't call either of those things,integration debt factors. That's regular development."*

Regarding branching factors such as branch depth and branch activity, the findings were pretty inconclusive as the developers do not necessarily use branches during development. This led to not being able to fully assess the impact of these two factors, even though several participants did believe that branch activity and depth can have an impact on integrating debt levels, it is not assured since not everyone works with branches. For instance, one participant said:

> *"Branches on top of each other can become messy very fast if you don't*

know what you're doing. There's definitely a risk that they do not rebase as often. But strictly speaking, no I don't think it's a factor.

In the interviews, the participants were asked about the integration debt factors gathered thus far. Figure 5.3 displays the interviewees' views on the gathered integration debt factors.

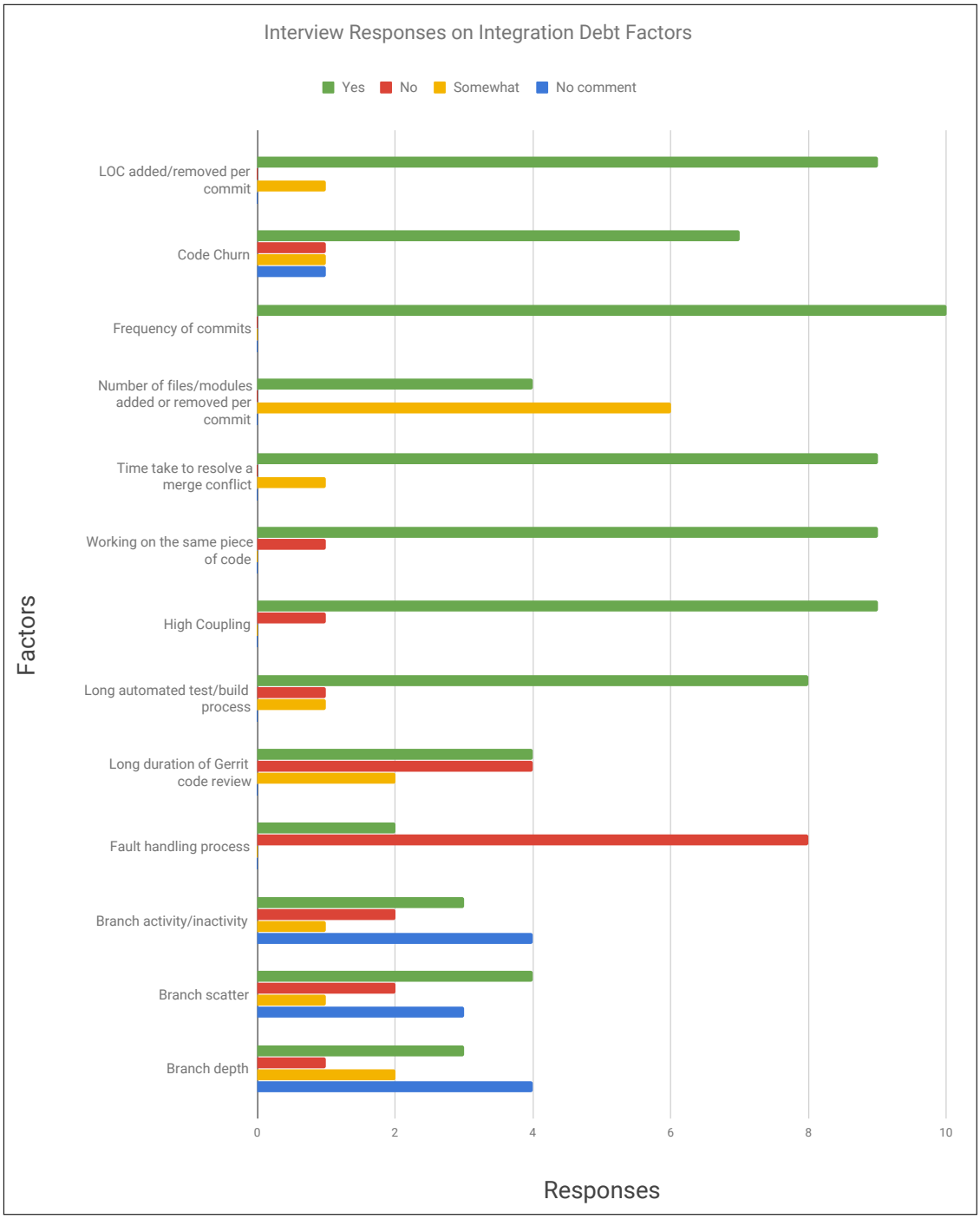


Figure 5.3: Interview responses about whether or not the factors gathered are actually integration debt factors

'Yes' means they definitely think it is a factor, 'No' that it is not a factor. The category of 'Somewhat' means that they thought it may be a factor but were not certain and 'No comment' simply means they did not have an answer. The raw data behind the figure is displayed in appendix A.2 where each participant was asked about their opinion of each of the factors and their impact on integration debt. The interview question itself did not contain the scale used in the figure, it was created afterward (during the analysis of the interview answers) to give an overview of the responses regarding the factors.

Having done the literature review, open-ended interviews with CI-managers and the interviews with developers, along with analysis of the interview data, a mapping of the different factors that had been gathered was created. The map included the different connections and relationships between the factors according to what had been said by CI-managers during the open-ended interviews, what the developers said in the semi-structured interviews, as well as what was found in the literature. If a relationship between factors (e.g. LOC affect merge conflicts) had been mentioned multiple times by multiple people or sources, it was considered for further investigation. The mapping shows where (literature, CI-manager interviews, developer interviews) each factor came from. Some factors have multiple sources of the factor indicated, this is because one method may only have partially suggested that it is a factor, and also what its potential relationships are with other factors.

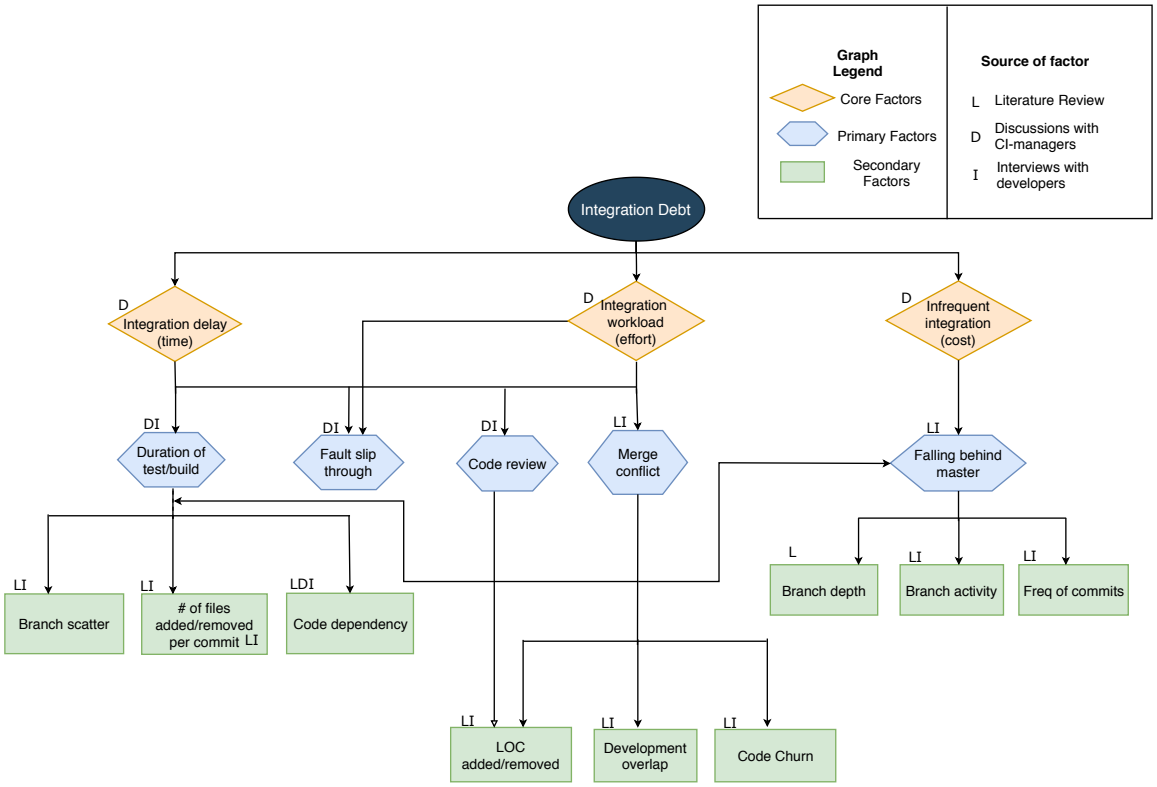


Figure 5.4: Integration debt factor mapping

Factor	**Sub-factors**
Core factor	***Primary factors***
Integration delay	Code review Duration of test/build Fault slip through Merge conflict
Integration workload	Fault slip through Merge conflict
Infrequent integration	Falling behind master
Primary factor	***Secondary factors***
Code review	Lines of code added/removed
Duration of test/build	Code dependency Branch scatter Number of files modified
Merge conflict	Code dependency Branch scatter Number of files modified Lines of code added/removed Development overlap Code churn Falling behind master (Primary factor)
Falling behind master	Branch depth Branch activity Frequency of commits

Table 5.2: Table of integration debt factors and their sub-factors

The map 5.4 has integration debt at the top and all sub-elements it points to are factors that it is caused by. As the map shows, there are different levels of factors, where each level refers to the number of steps it is from integration debt at the top. To further clarify the connections between the factors, in case the mapping figure is unclear, the sub-factors for each factor can also be seen in table 5.2. On the first level are the core factors that make up integration debt, on the next level are the primary factors that directly cause the core factors. Finally, there are the secondary factors that affect or cause the primary factors. It is mainly the secondary factors that the metrics will focus on because they are the root causes (within our scope) of integration debt, and they are also measurable within the CI-system.

The factor map 5.4 is not exhaustive in that it does not include all possible factors since some were out of scope, some were put together (e.g. availability of CI resources is a factor that is part of CI-system factors) and some may have been undiscovered in this study. The factor map is also not necessarily applicable for all situations in the sense that a secondary factor may generally cause a primary factor but not necessarily always. For instance, LOC may generally affect merge conflict

risk (according to the developers and literature) but in a special case, for example, if a file is simply moved and thus counts as added LOC, it may not affect merge conflict risk.

In another example, falling behind master contributes to merge conflicts and causes increased workload (effort) and insufficient integration (cost) in the general case. In a special case, it may be that falling behind master is the preferable thing to do, for example, if a system part needs to be completed before integration, and thus it may reduce the effort and cost compared to not falling behind master. Even in the special case, there is an integration debt building up but it is a deliberate choice that reduces the risk of integration problems, contrary to what the map would suggest for the general case. The factor map should thus be viewed as a generalized model of integration debt factors rather than a complete and definite mapping.

There were several assumptions that were made at this point before the data mining. These assumptions were based on the information gathered so far, especially from the interviews, as well as our current understanding of the domain. It was necessary to make some assumptions about the factors and relationships in order to limit the amount of statistical analysis needed.

The potential correlations investigated were based on the mapping of the integration debt factors (see figure 5.4) that had been created. We assumed that these correlations were the interesting ones to investigate, specifically the relationships between the primary and secondary factors. This is because the secondary factors are assumed to cause or increase the risk of the primary factors, which in turn are the ones that are directly connected to the core factors which is what integration debt is or results in. This makes it interesting to measure the secondary factors because measuring the primary factors directly means that the integration problem has already happened. For example, merge conflicts occurrences could be measured or counted but then they would already have happened, whereas measuring the secondary factors connected to merge conflicts would allow for merge conflict risk to be measured or estimated before merge conflicts happen.

The supposition going into the data mining and subsequent data analysis was that each secondary factor was correlated with the primary factor that it is connected to in the map. For instance that the LOC added or removed is correlated with occurrences of merge conflicts. Through the data mining, these relationships were investigated to determine if they are actually present or not.

5.4 Data mining & questionnaire results

The purpose of repository mining is to statistically investigate each secondary factor against its associated primary factor, shown in the factor mapping figure (5.4). Repository mining will allow the investigation of correlations between primary and

secondary factors. This means investigating if the secondary factors that have been defined really are causes of merge conflicts and longer duration of test/build as has been indicated by the results from the interviews and literature review. The data mined came from a six month period of development at the company which includes all commits in that time that were committed to the repository of one of their main projects.

For the ease of the reader, the questionnaire results for each factor are also brought up in this section, alongside the statistical result for each factor. In this way, the reader can get an insight into both the statistical result and the questionnaire result for the factors. The questionnaire was conducted at the late stages of data mining where its purpose was to validate the findings (from the literature review and interviews) regarding affecting factors in integration debt. It consisted of 19 questions which were sent out to all developers online, via email. In the end, a total of 30 participants took part in the questionnaire. The template of the questionnaire is displayed in the appendix A.5. Fifteen out of nineteen questions in the questionnaire were multiple choice questions using the Likert scale which allowed for better categorization of findings. There were four optional questions that required a participant to write a text and express their thoughts on any missed part of integration debt. These findings are later on discussed in the discussion section.

In the subsections below, the factors are investigated using CI data and the questionnaire results for each factor are also presented. There are some factors that were not investigated through data mining but only investigated using the qualitative methods of interviews and a questionnaire. These will be explained, and their results presented, in the section for qualitatively investigated factors.

5.4.1 Lines of code

The first factor that is being investigated is lines of code. Developers submit their code to the repository through commits and one way of measuring the size of commits is by counting their lines of code added and removed. It became clear from literature [19] that, a high number of LOC per commit could increase the likelihood of merge conflicts and longer duration of test/build during CI development. Additionally, interview results also confirmed the involvement of LOC as a factor on merge conflict and duration of test/ build. Therefore these two assumptions are examined in the two sub-sections below (5.4.1.1 and 5.4.1.2), using the data from the repository.

5.4.1.1 LOC versus merge conflict

The data for LOC was grouped on a weekly basis and for each week the sum of LOC added/removed and the total number of merge conflicts were compared for

each week of development. The assumption generated from literature review and interview, suggests an increase in LOC would have an impact on the occurrence of merge conflicts. This section investigates the correlation between both LOC removed and added against the total occurrence of merge conflicts for each week of development. The repository data was gathered using a Java program which is available in appendix A.3. The script allowed fetching the lines of code of commits from the commit using their IDs. Their LOC was added to the sum of LOC for that week.

The results of the LOC against merge conflict were plotted on a line graph shown in figure 5.5. The number of merge conflicts for each week is shown in blue color and LOC added and removed are respectively in red and yellow colors. In the figure, the horizontal X-axis represents the number of the week. The left Y-axis represents the total number of LOC while the right-most Y-axis shows the total occurrence of merge conflicts for that week. The graph suggests that as the lines of code increases, the likelihood of merge conflicts also increases. Although this correlation seems more accurate for LOC added compared to LOC removed. The graph visualizes a moderate correlation, especially with LOC added and merge conflict, in comparison with LOC removed which seems to have a slightly less correlation with merge conflicts. By looking at the graph (5.5), it becomes apparent that several weeks had a very large number of total LOC. After further investigation, it was discovered that often when production was at its peak, the incoming LOC was very high, for example from weeks 11 to 14 when was the last weeks of the year, the total LOC became very large. These outliers are however still considered in the statistical calculation of correlation as they were considered valid data points.

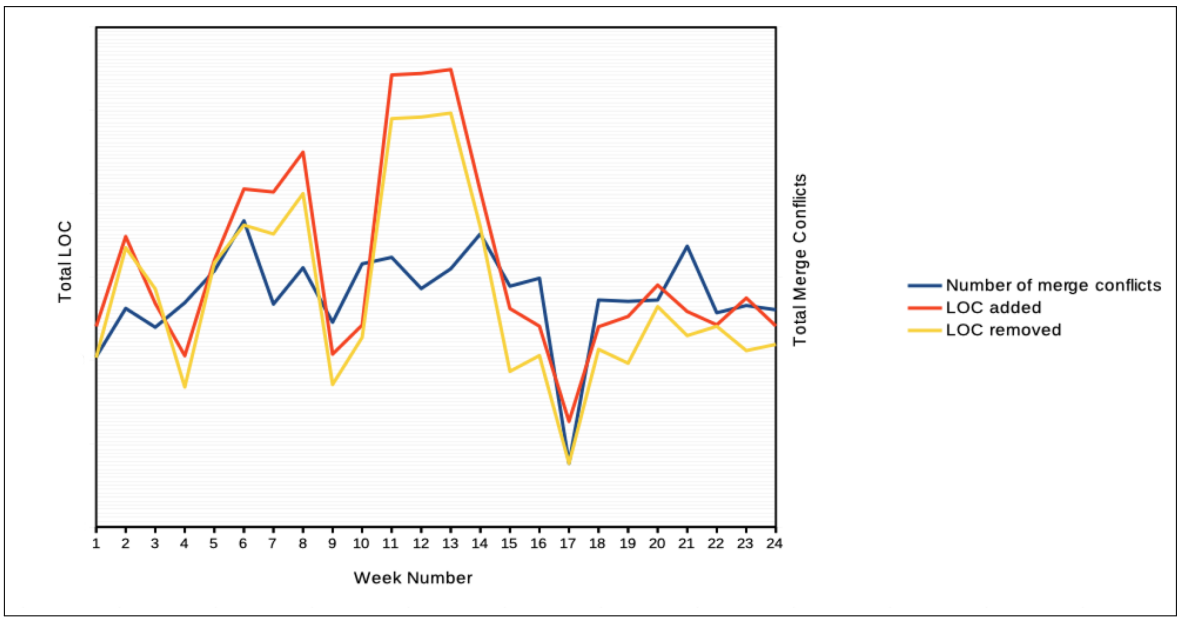


Figure 5.5: Total LOC added & removed against occurrence of merge conflicts - including outliers (note that the y-axis values have been redacted at the request of the company involved in the study, however the pattern, and thereby the correlation is still visible)

To statistically analyze the raw results of LOC added/removed against the occurrence of merge conflicts, the data needed to be tested for normality to see if the data is normally distributed. By looking at graph 5.5, it can be visually confirmed that data are not normally distributed. To validate this, the Shapiro-Wilk normality test was used and the p-value was extremely below the alpha level (0.05) which rejects the null hypothesis of data being normally distributed. Thus it can be concluded that LOC data is not normally distributed and so a non-parametric test needs to be used in investigating the association between merge conflicts and LOC. Spearman's correlation coefficient (rho) was used and the result is shown below. For both LOC added and removed, the significance of the correlation value was also tested using a T-test and its p-value is also shown below.

$$\begin{aligned} Shapiro\ Wilk's\ \ p-value\ for\ LOC\ added\ &= 0.00000001719 \\ Spearman's\ rho\ for\ LOC\ added\ &= 0.5579474 \\ T\ test's\ p-value\ for\ LOC\ added &= 0.00461 \end{aligned}$$

$$\begin{aligned} Shapiro\ Wilk's\ \ p-value\ for\ LOC\ removed\ &= 0.00000002062 \\ Spearman's\ rho\ for\ LOC\ removed\ &= 0.5327245 \\ T\ test's\ p-value\ for\ LOC\ removed &= 0.00736 \end{aligned}$$

As can be seen for both LOC added and removed, there is a moderately strong correlation (according to the guideline used as mentioned in the methods section) with merge conflicts. Both results are also statistically significant as the p-value is lower than the significance level of 0.05.

The result from the questionnaire regarding LOC's relationship with merge conflicts also shows that the developers overwhelmingly agreed that higher LOC increases the risk of merge conflicts.

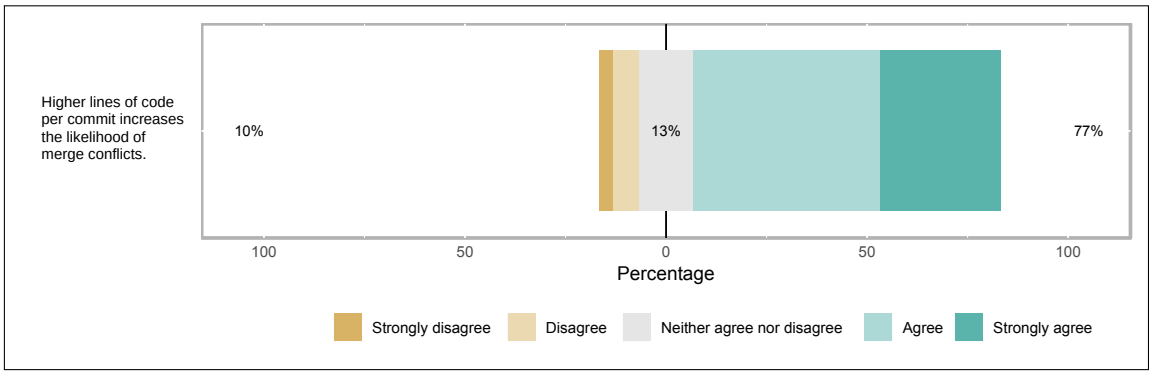


Figure 5.6: Questionnaire responses regarding LOC increasing risk of merge conflicts

77% of the developers either agreed or strongly agreed that merge conflict risk

increases with LOC and only 10% disagreed or strongly disagreed. The remaining 13% neither agreed nor disagreed. This result further confirms that LOC is indeed an important factor in integration debt and that LOC does affect merge conflict risk.

5.4.1.2 LOC versus duration of test and build

As previously mentioned in the section for the company's CI flow, the company uses a dedicated CI machinery to test the quality of the code before it gets merged to the mainline. The assumption generated from literature review and interview, suggests an increase in LOC would have an impact on the duration of CI's test and build. This section investigates the correlation between both LOC removed and added against the total duration of automated test and build. Data regarding the duration of the test and build for all commits was gathered from CI-system's historical data. The fetched time duration was then added together and sorted in weekly groups. The data contains the total duration of test & build which allows comparison of this number against LOC added and removed for each week. The total duration of test and build refers to the entire amount of time taken for all commits in each week of development.

Figure 5.7 displays the correlation of lines of code added and removed against the total duration of automated test and build. The horizontal X-axis in the graph, indicates the number of weeks, while the left side Y-axis is the total LOC duration of test and build. As discovered in the previous section, LOC data appear to not be normally distributed as indicated by Shapiro Wilk's p-value. In parts of this graph, there is some visible correlation, especially with LOC added compared to LOC removed. Although it cannot be visually confirmed and statistical correlation analysis is required.

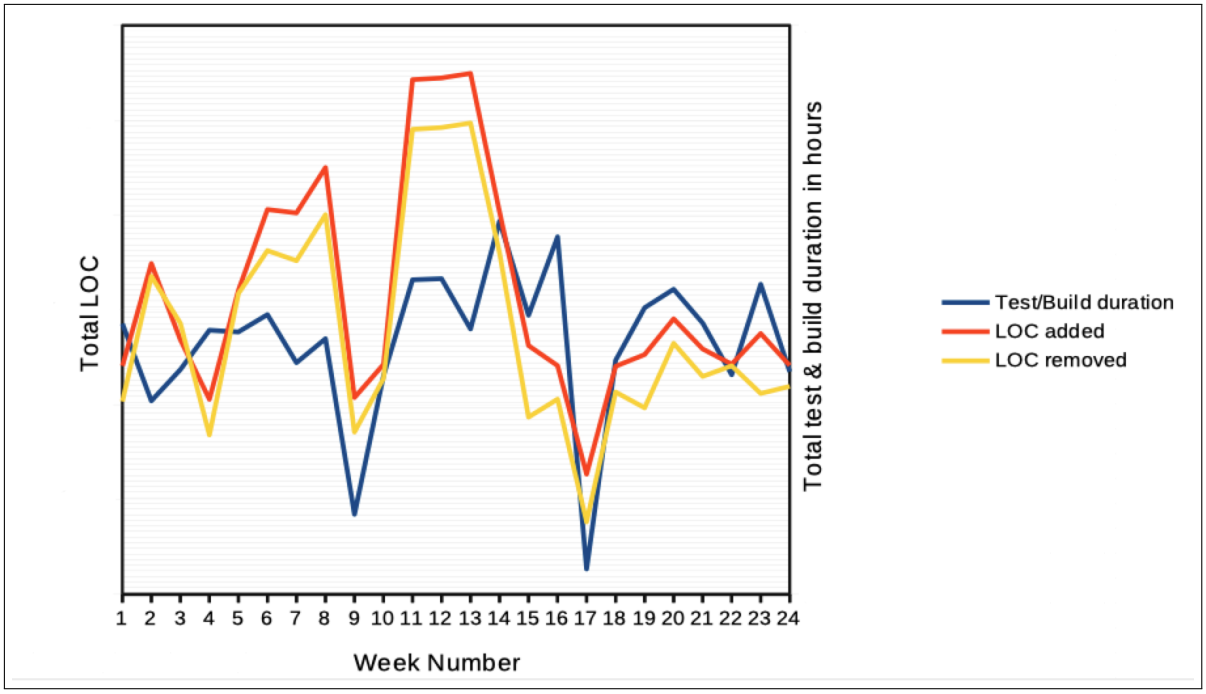


Figure 5.7: Weekly data of total LOC added or removed and their correlation with duration of automated test & build (note that the y-axis values have been redacted at the request of the company involved in the study, however the pattern, and thereby the correlation is still visible)

Since it was indicated earlier that LOC data is not normally distributed and there are outliers in the data set, Spearman's rank-order correlation is used to calculate the relationship between LOC and duration of test and build. The calculated coefficient values of Spearman's are shown below together with their T test's p-values.

$$Spearman's\ rho\ for\ LOC\ added\ = 0.4295652$$
$$T\ test's\ p-value\ for\ LOC\ added = 0.03723$$

$$Spearman's\ rho\ for\ LOC\ removed\ = 0.2591304$$
$$T\ test's\ p-value\ for\ LOC\ removed = 0.2205$$

The result indicates a moderate relationship for LOC added with the duration of the test and build, while the rho value for commits with LOC removed shows a weaker relationship. The T test's p-value for LOC added also indicates a statistically significant result (with significance level 0.05) but the T test's p-value for LOC removed does not.

As for the questionnaire responses from the developers regarding this correlation, they generally did not think that LOC increases the duration of test and build.

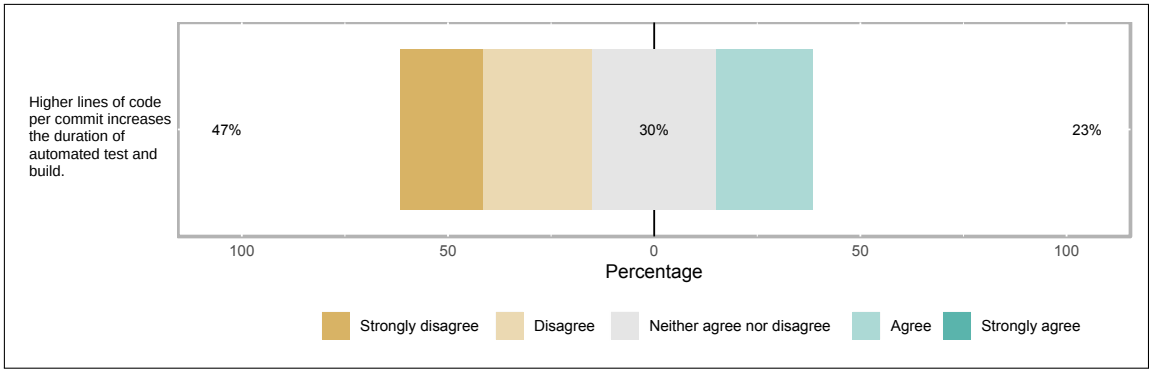


Figure 5.8: Questionnaire responses for if LOC increases test/build duration

As can be seen in figure 5.8, only 23% of the developers agreed that LOC increases test and build duration, with none of them strongly agreeing. 47% either disagreed or strongly disagreed. These results show that the impact LOC has on test/build duration is not particularly large, and it is significantly smaller than its impact on merge conflict risk.

5.4.2 Code Churn

Code churn is defined as the absolute value of the code delta [18]. Meaning the sum of the number of lines of code added plus the sum of the number of lines removed provides the value for code churn. Code churn was investigated as it was suggested in literature [18] that it affects the rate at which defects may be introduced, and thereby increases the risk of integration problems, namely merge conflicts. Code churn is a very similar factor to LOC, and the same assumptions are made, i.e. that a higher amount will increase the risk of merge conflicts.

Spearman's was used instead of Pearson's because as for LOC there were a few significant outliers that were investigated and determined to be valid data points. The Spearman's correlation coefficient was calculated and the statistical significance was tested:

$$\begin{aligned} Spearman's\ rho &= 0.5596869 \\ T\ test's\ p-value\ &= 0.00457 \end{aligned}$$

The correlation value does suggest that code churn is related to merge conflicts and that there is a moderately strong correlation. This shows that the assumption was correct and that it is a relevant integration debt factor that affects merge conflict risk. The p-value is also lower than the significance level (of 0.05) which indicates a statistically significant result.

From the questionnaire responses (see figure 5.9), we can see that the developers clearly thought that higher code churn increases merge conflict risk.

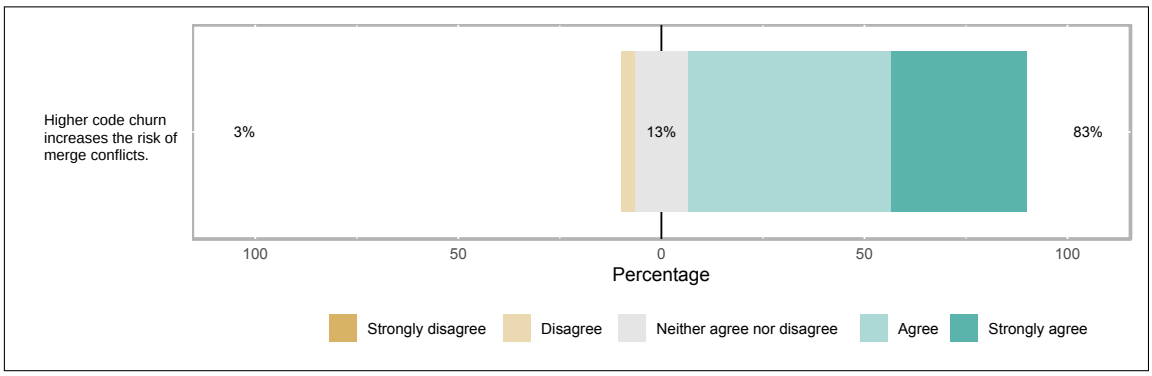


Figure 5.9: Questionnaire responses for code churn increasing merge conflict risk

83% of the developers agreed or strongly agreed that code churn increases the risk of merge conflicts. Only 3% disagreed. Building on the statistical result regarding the correlation, this shows that code churn definitely is an integration debt factor.

5.4.3 Number of files

Results from the literature review and the interviews suggested that the number of files (NOF) modified is potentially correlated with both merge conflict occurrences and test/build duration. This was investigated through the data from the repositories and the CI-system. In the two subsections below (section 5.4.1.1 and section 5.4.1.2), the correlation of the number of files modified per commit is calculated against merge conflict and duration of automated test and build.

5.4.3.1 Number of files versus merge conflict

In investigating the relationship between NOF and occurrence of merge conflicts, i.e. the number of files modified compared against merge conflicts occurrences on a weekly basis. The idea was to see whether an increase in NOF would have an impact on occurrence of merge conflicts. Thus, 24 weeks of data was selected and for each week the total number of files modified was added together to get the total NOF of that week of development.

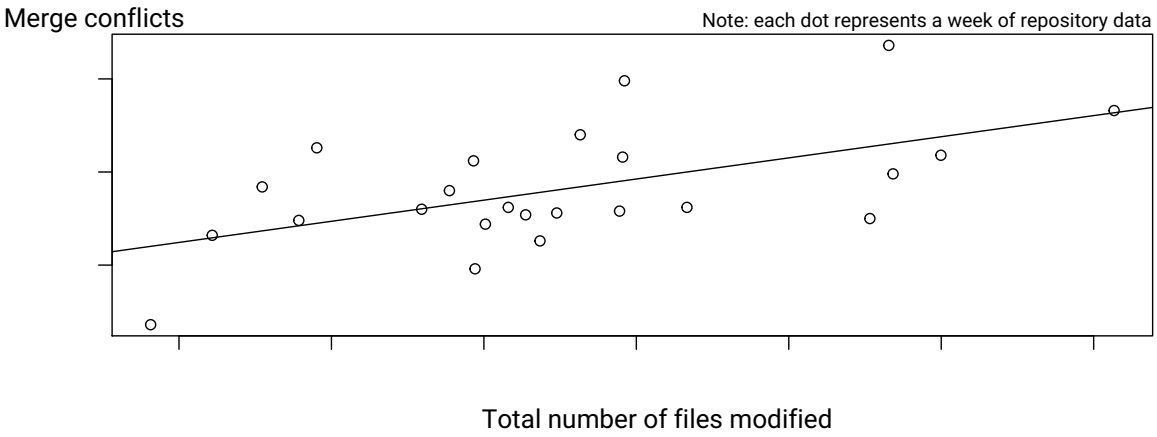


Figure 5.10: Linear regression model of merge conflicts and number of files modified per week (note that the values have been redacted at the request of the company involved in the study, the trend of the data is however still visible in the graph)

Pearson's correlation coefficient was used to measure the correlation between the number of files modified and merge conflicts. Since Pearson's assumes a normal distribution, a Shapiro Wilk normality test was run on the data to ensure normal distribution. The result indicates a normal distribution of data as the p-value is significantly higher than the alpha (0.05). The p-value for the Pearson correlation coefficient itself was also calculated using a significance test (t-test).

$$\begin{aligned} Shapiro\ Wilk's\ \ p-value &= 0.4494 \\ Pearson's\ r-value\ \ &= 0.5643074 \\ T\ test's\ p-value\ \ &= 0.004073 \end{aligned}$$

The result from calculating Pearson's correlation coefficient shows a moderately strong relationship between the number of files modified and merge conflicts. The p-value indicates a statistically significant result for significance level 0.05.

The questionnaire responses (figure 5.11) showed the developers agreeing that merge conflict risk increases with the number of files modified. 83% agreed or strongly agreed that the number of files increases merge conflict risk, whereas only 10% disagreed. 7% neither agreed nor disagreed.

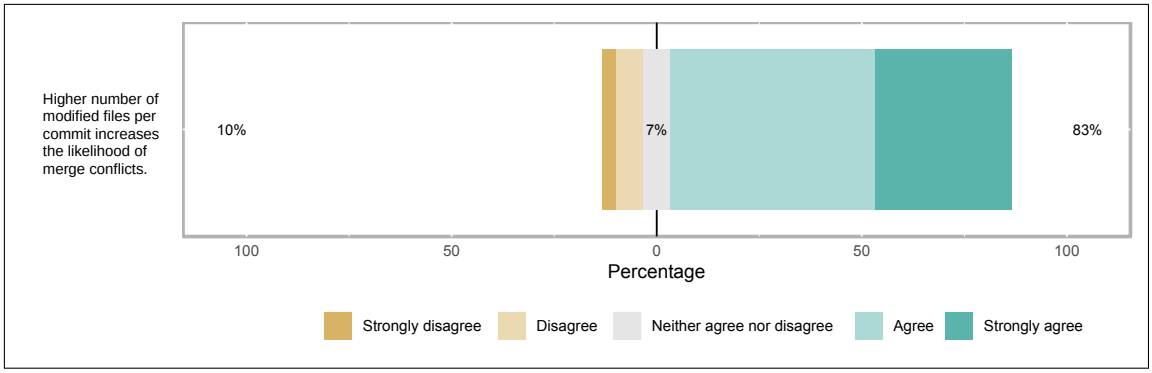


Figure 5.11: Number of files modified versus merge conflicts - questionnaire responses

These results show that there is a correlation between the number of files modified and merge conflicts, and that number of files is a relevant factor to measure for integration debt.

5.4.3.2 Number of files versus duration of automated test and build

The data used to analyze NOF versus duration of automated test and build includes the total duration of automated test and build. This data which was fetched from the CI-system can be compared against the number of files modified on a weekly basis, to examine the relationship NOF against the duration of test/build. The results are plotted on the linear regression model below 5.12 where the X-axis is the total number of files modified and the Y-axis is the total time it took for commits in each week to go through the CI's automate test and build phase.

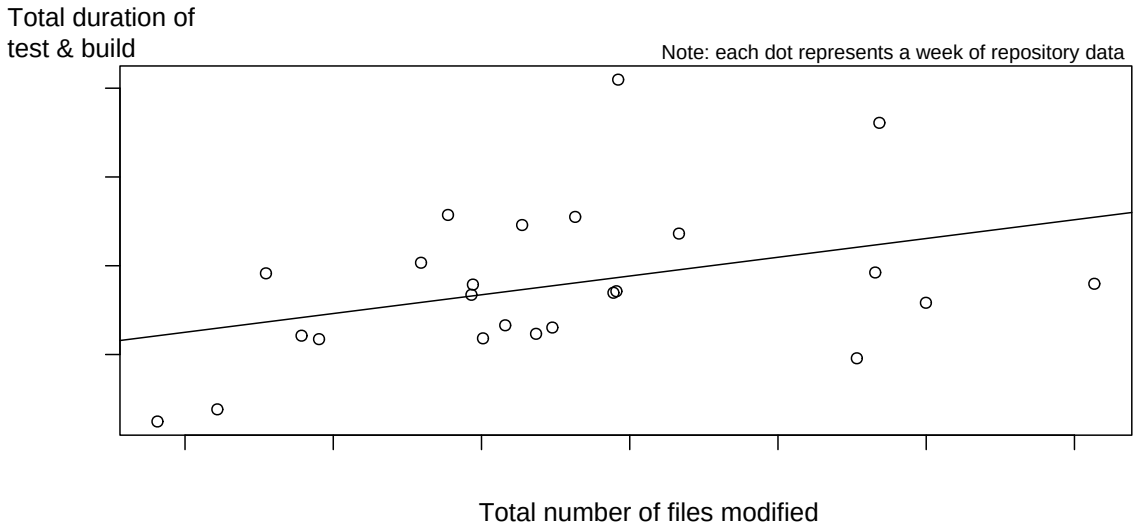


Figure 5.12: Linear regression model of total number of files added against total duration of test and build for each week of development (note that the values have been redacted at the request of the company involved in the study, the trend of the data is however still visible in the graph)

$$\begin{aligned} Shapiro\ Wilk's\ \ p-value &= 0.1349 \\ Pearson's\ r-value\ \ &= 0.3839306 \\ T\ test's\ p-value\ \ &= 0.064 \end{aligned}$$

The results from Shapiro Wilk above shows that the data is normally distributed ($p > 0.05$), meaning the correlation of NOF against the duration of test and build can be measured using a parametric test such as Pearson's correlation coefficient. The calculated r value is 0.38 which points to a fairly weak relationship between the variables. This means that there is a weaker correlation between the number of files modified and the duration of test and build. The p-value is higher than the significance level 0.05 which means that we can not consider it to be a statistically significant result.

The developers did not think that the number of files modified increasing test/build duration (figure 5.13) is as significant of a correlation as the number of files increasing merge conflict risk as seen in figure 5.11.

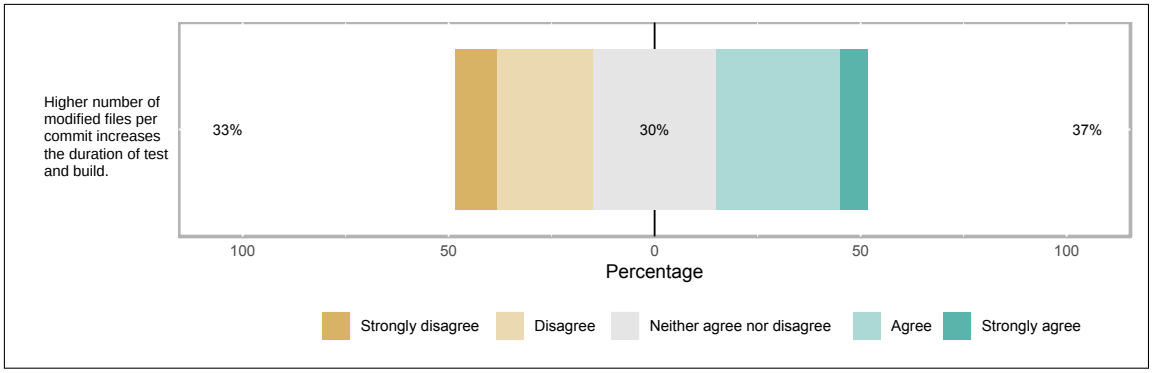


Figure 5.13: Questionnaire responses for number of files modified versus test/build duration

37% of the developers agreed or strongly agreed that the number of files increases test and build duration, whilst 33% disagreed or strongly disagreed. Looking at all of the results for the number of files modified, it appears that it is indeed an integration debt factor, and it affects merge conflict risk more than test and build duration.

5.4.4 Frequency of commits

The frequency of commits is a secondary factor that, in the factor mapping, goes under the primary factor of falling behind master which in turn connects to merge conflicts. To statistically investigate the frequency of commits as an integration debt factor, it is thus compared against merge conflicts.

The frequency of commits is defined as the total number of commits in a certain time period, which in this case is weekly. Literature [20], suggested that it might be a factor in integration debt since commits in temporal proximity were shown to cause problems, and it was also mentioned in the interviews as an obvious factor for integration debt. The assumption was made that the primary factor it affects (other than falling behind master) is merge conflicts, i.e. the more commits there are, the more merge conflicts there should be.

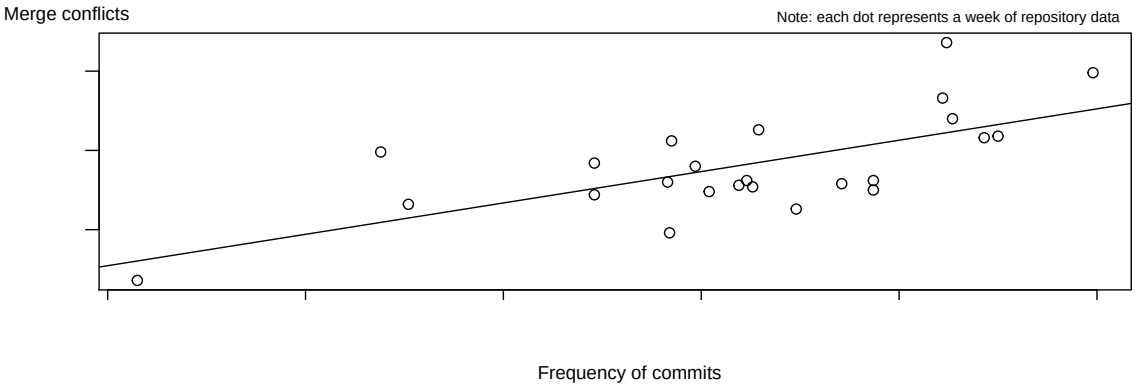


Figure 5.14: Linear regression model of merge conflicts and total commits per week (note that the values have been redacted at the request of the company involved in the study, the trend of the data is however still visible in the graph)

The CI-system and repository data were categorized by week. For each week, the total number of commits was compared against the total number of merge conflicts. The data can be seen in plot 5.14. The normality of the data was tested with a Shapiro-Wilk normality test:

$$Shapiro\ Wilk's\ \ p - value = 0.1447$$

Then the correlation between the number of commits in the selected weekly time-span (i.e frequency) and merge conflicts was calculated using Pearson's correlation coefficient and a significance test (t-test) was performed to calculate the p-value:

$$Pearson's\ r = 0.6800936$$
$$T\ test's\ p - value = 0.0002559$$

As we can see, there is a strong correlation between the frequency of commits and merge conflicts. This suggests that the frequency of commits is an integration debt factor that should be measured in order to determine integration debt levels since it has a significant effect on merge conflict risk. The p-value is below the significance level of 0.05 which allows to reject the null hypothesis (of statistical insignificance) and consider it a statistically significant result.

In the questionnaire, the developers overwhelmingly agreed that a low frequency of commits increases the risk of merge conflicts because of falling behind the master branch.

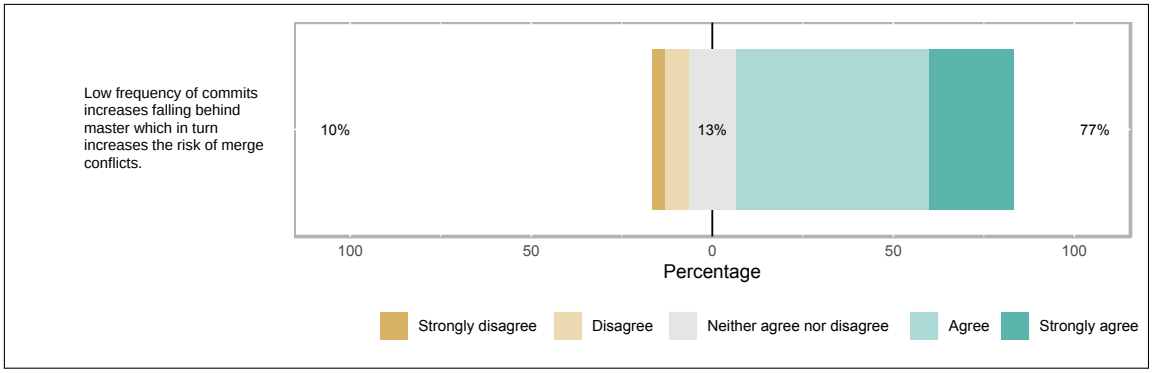


Figure 5.15: Questionnaire responses for frequency of commits

As can be seen in figure 5.15, 77% of the developers either agreed or strongly agreed that low frequency leads to an increased risk of merge conflicts. 13% neither agreed nor disagreed and only 10% disagreed or strongly disagreed. This combined with the strong statistical correlation shows that the frequency of commits is an important integration debt factor.

5.4.5 Branch Scatter

Branch Scatter refers to how spread out the development is across different branches. In the two sub-subsections below, branch scatter (i.e.the number of branches in the project) is compared against the occurrence of merge conflicts and the duration of automated test & build.

5.4.5.1 Branch scatter versus merge conflict

When conducting the literature review, it was discovered that high branch scatter has been shown to increase faults during development [21]. The assumption being made is that the faults (which are affected by branch scatter) increase the risk of merge conflicts and the duration of testing and building. In the results below, we measure the impact of branch scatter on causing merge conflicts. To do that, all the branches that were active during 6-month span of development were fetched and their commits were counted. These branches consist of local, personal and team branches. For better analysis, the data were broken down into weekly data. The data from the branches that were active in development per week can be seen in figure 5.16.

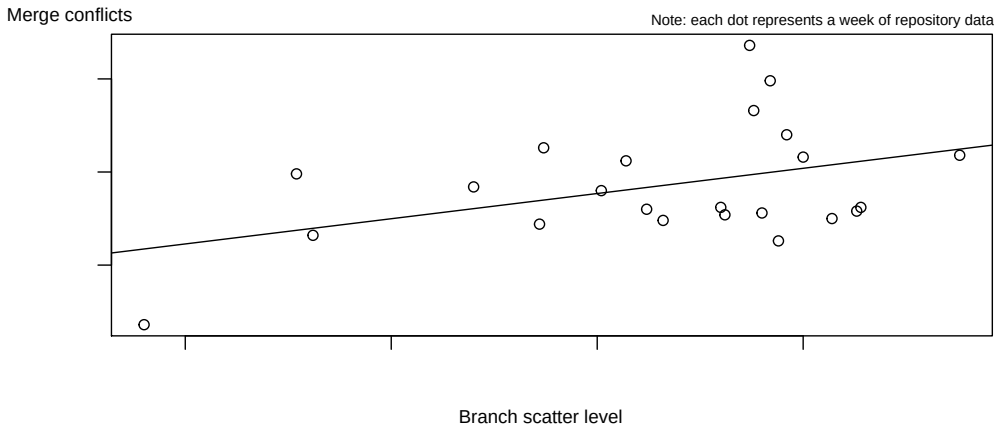


Figure 5.16: Branch scatter against merge conflict - weekly data (note that the values have been redacted at the request of the company involved in the study, the trend of the data is however still visible in the graph)

To check if the weekly data is normally distributed, the Shapiro Wilk formula was used. The p-value indicates that the data is not normally distributed. Therefore a non-parametric test that does not assume normal distribution needed to be used for investigating the association degree between branch scatter and merge conflicts.

$$Shapiro\ Wilk's\ \ p-value\ = 0.0317$$

The correlation value used was Spearman's and its p-value was also calculated:

$$Spearman's\ rho = 0.2552729$$
$$T\ test's\ p-value = 0.2286$$

The correlation value is very low and suggests that branch scatter is not related to merge conflicts and can thus be disregarded as such. The p-value is above the significance level which disallows us from considering it as a statistically significant result.

While the statistical correlation was very weak, the responses (figure 5.17) from the developer questionnaire generally agreed that branch scatter does increase the likelihood of merge conflicts.

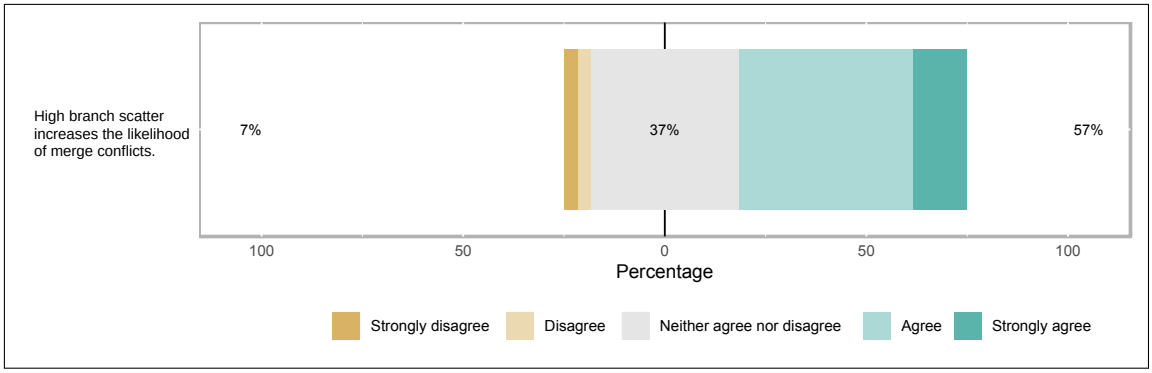


Figure 5.17: Questionnaire responses for branch scatter increasing merge conflict risk

Only 7% of developers disagreed, or strongly disagreed, that branch scatter increases the risk of merge conflicts while 57% agreed or strongly agreed. This shows that despite branch scatter having a very weak statistical correlation with merge conflicts, it may still be of some interest as an integration debt factor since the developers seem to regard it as a potential factor.

5.4.5.2 Branch scatter versus duration of automated test and build

In addition to its potential impact on merge conflicts, the impact of branch scatter against the duration of automated test and build is measured and investigated statistically.

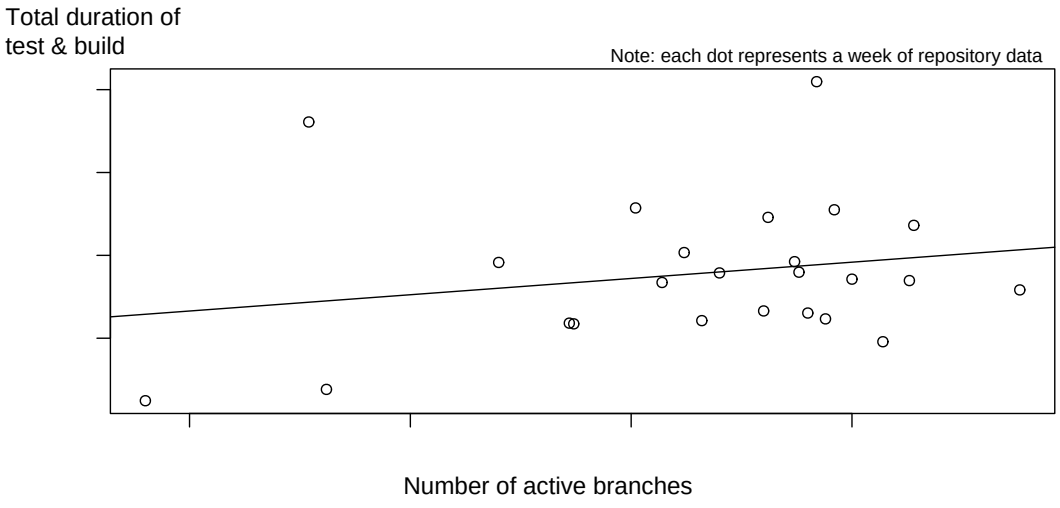


Figure 5.18: Branch scatter against duration of test & build - weekly data (note that the values have been redacted at the request of the company involved in the study, the trend of the data is however still visible in the graph)

The correlation value (and its p-value) for branch scatter versus the duration of test and build was the following:

$$Spearman's\ rho = 0.1652174$$
$$T\ test's\ p - value = 0.4387$$

The correlation value is again very low, which shows that branch scatter is not related to the duration of test and build. According to these results, branch scatter is not a significant integration debt factor.

In the questionnaire, most developers did not agree that branch scatter increases the duration of test and build. Only 30% did so, whereas 40% disagreed (or strongly disagreed) and 30% neither agreed nor disagreed. This, combined with the very weak statistical correlation, shows that the relationship between branch scatter and test/build duration is not worth considering, and that branch scatter is not a significant integration debt factor.

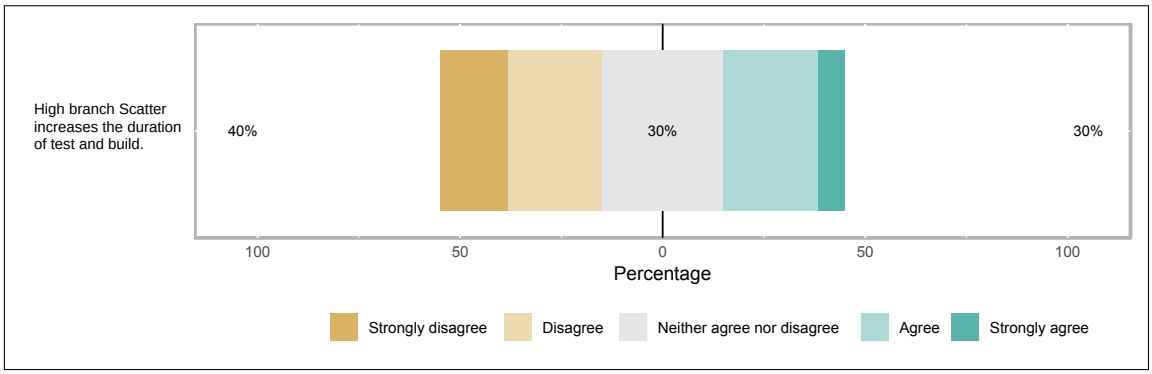


Figure 5.19: Questionnaire responses for branch scatter increasing duration of test and build

5.5 Qualitatively investigated factors

There are a number of potential integration debt factors that were discovered in the literature review and from interviews with developers and CI-managers that were not part of the data mining and analysis. The reasons were that they were unable to be defined and/or measured in a satisfactory way within the constraints of the company's CI-system and code repository. A short description and an explanation for each unmeasured factor are provided below.

5.5.1 Branch activity

Branch activity had been shown in literature [21], to increase defects. During the interviews, some developers also thought it may be an integration debt factor. It was left unmeasured since it overlapped heavily with the frequency of commits and a suitable definition or re-definition for it (and how to measure it) was not found.

The questionnaire result (figure 5.20) for branch activity shows that none of the developers disagreed that low branch activity increases falling behind master, which, in turn, increases the risk of merge conflicts. Branch activity would thus be an interesting factor to investigate further in future research.

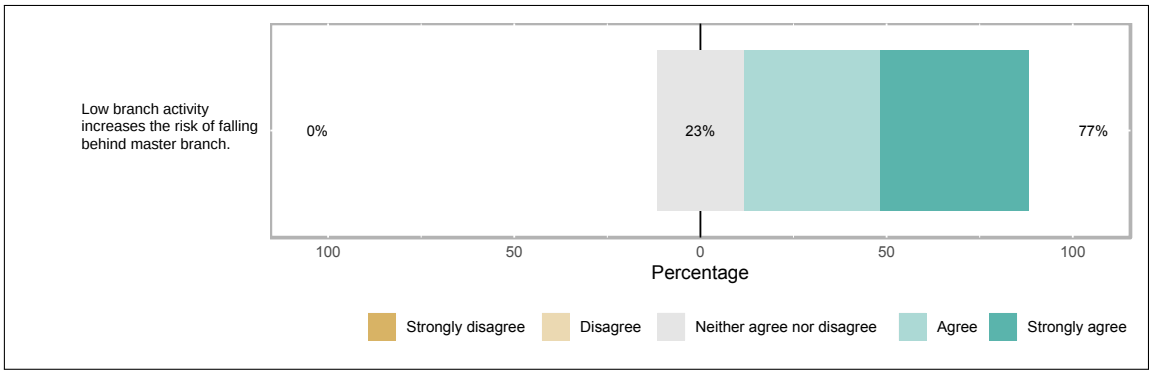


Figure 5.20: Questionnaire results regarding branch activity

5.5.2 Duration of Gerrit code review

During the open-ended interview with CI-managers and the developer interviews, the duration of code reviews was brought up as a potential integration debt factor since it presumably affects how long the integration takes from the code being pushed to being merged. In the factor mapping, it was determined to be a primary factor since it has other secondary factors, such as LOC, that affects it. The reason it was not statistically measured or analyzed was that there was no available data on the duration of code reviews at the company, as such it could not be measured.

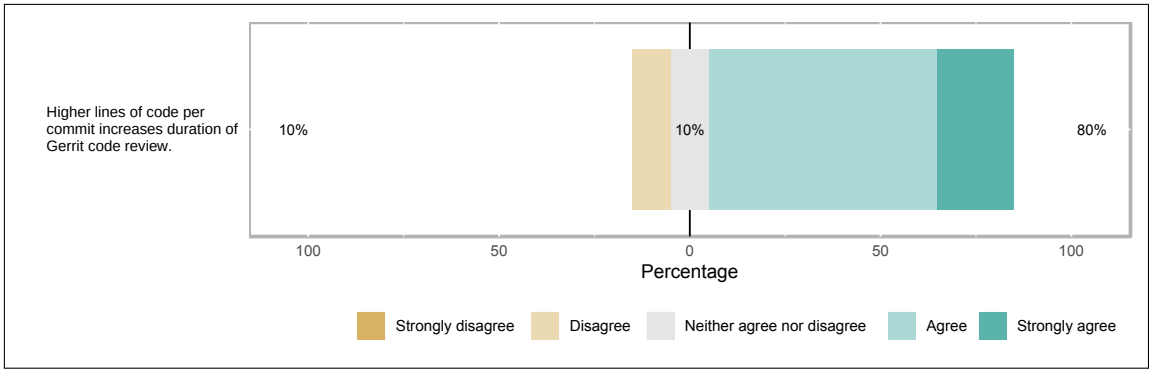


Figure 5.21: Questionnaire responses regarding LOC increasing duration of code review

5.5.3 Fault slip through

Fault slip through was brought up by CI-managers and by developers during the interviews as a potential integration debt factor since it may cause problems that increase integration workload or delay integration. Again, there was no available data on this so it could not be measured.

From the questionnaire result for this factor, we can see that 93% of the developers thought that fault slip throughs increase the risk of integration problems. None of the developers disagreed. This shows that fault slip through would be an interesting factor to further investigate.

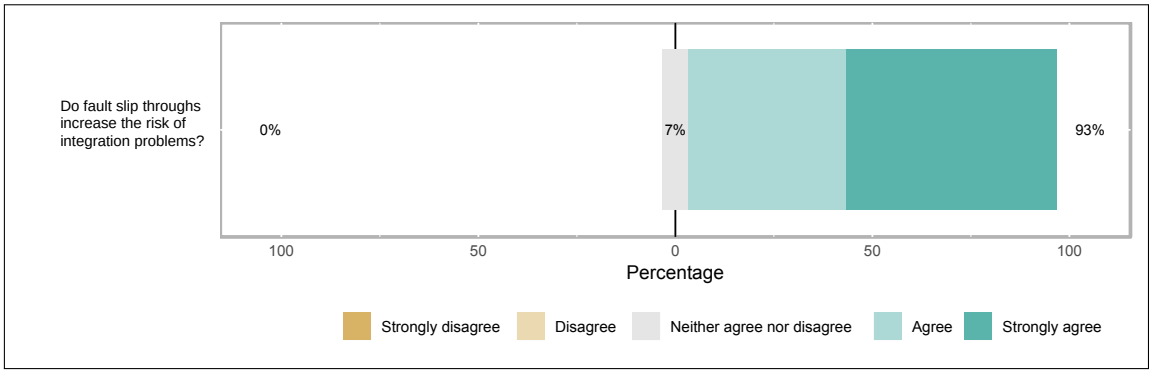


Figure 5.22: Questionnaire responses regarding fault slip through

5.5.4 Branch depth

Branch depth was mentioned in literature [21] and during the interviews, some developers also thought it may be an integration debt factor. The assumption was that the further away you are from the master branch, the greater the risk

of merge conflicts. Due to the workflow at the company, the branch depth could not be reliably measured. It was difficult to distinguish individual branches in the repository and gather data from them because most developers at the company did not use branches.

70% of the developers who took part in the questionnaire agreed or strongly agreed, that high branch depth increases the risk of falling behind master, which in turn increases merge conflict risk. This does show that branch depth is a potential integration debt factor.

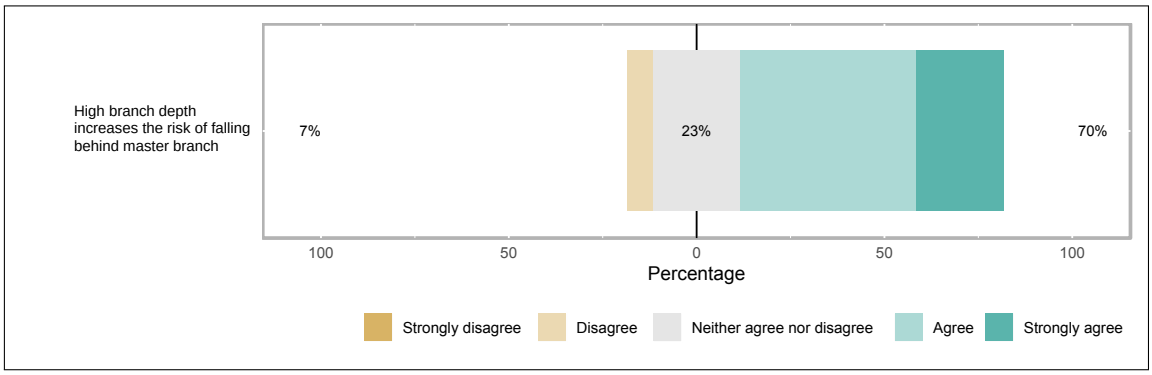


Figure 5.23: Questionnaire responses regarding branch depth

5.5.5 Code dependency between commits

Dependency was brought up in literature, CI-manager interviews and developer interviews as a factor that could be important to integration debt. The assumption is that it affects merge conflict risk since highly interdependent code is more likely to cause a conflict with something else. It will also affect test/build duration since dependency is (according to the CI-managers at the company) what triggers many of the tests during the test/build phase.

The information on how dependency is determined by the CI-system was not available. It was also not clear how to reliably measure it within the CI-system and code repositories. As such it was left unmeasured.

As can be seen in figure 5.24 and figure 5.25, the developers generally agreed that dependency increases both the likelihood of merge conflicts and the duration of test/build. It shows that dependency is one of the more important factors to consider in the domain of integration debt.

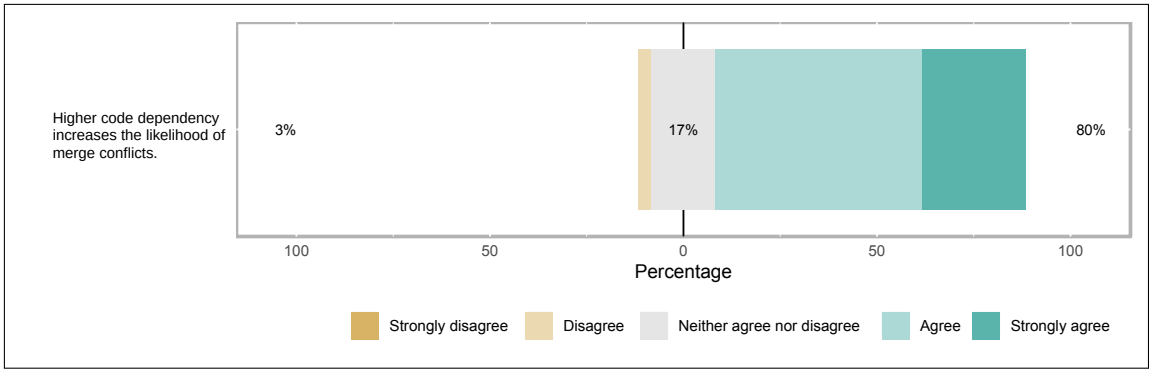


Figure 5.24: Questionnaire responses regarding dependency versus merge conflicts

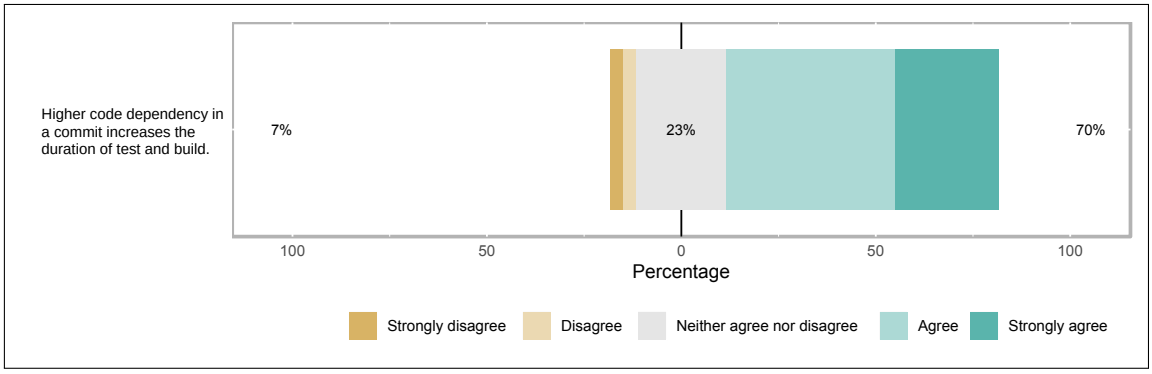


Figure 5.25: Questionnaire responses regarding dependency versus test/build duration

5.5.6 Code development Overlap

From the literature review, development overlap was found to be a potential factor. The definition used by [20] was commits that are pushed in close temporal proximity. There were a few problems with measuring development overlap which is why it was left unmeasured.

Firstly, the time-span for what counts as temporal proximity was unclear and there were multiple different potential time spans that would likely have yielded different results. Secondly, there was a consideration to be made for the origin of commits that are counted as overlapping, i.e. should it be from different personal branches, authors, team branches, or perhaps it should include the commits from the same origin? These issues made it overly time-consuming to investigate and this factor was thus left unmeasured.

The questionnaire result indicates that development overlap is indeed a factor worth considering as a part of integration debt since 90% of the developers agreed (most of them strongly agreed) that development overlap increases the risk of merge conflicts.

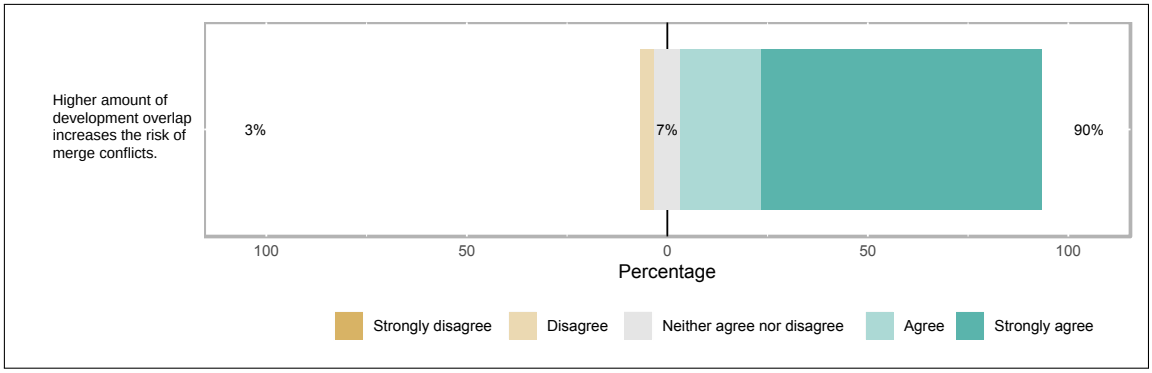


Figure 5.26: Questionnaire responses regarding development overlap

5.6 Suggested threshold for LOC metric

So far in this study, the potential factors involved in integration debt have been investigated through literature review, interview, statistically analyzing repository data and a questionnaire. The findings are helpful in determining the factors involved in integration debt. To capture the impact of the integration debt factors and be able to predict integration debt levels based on their metrics, a set of target values or as they will be referred to, *thresholds*, are needed to be created. Through repository data, these thresholds can be extracted to show the debt level of unintegrated code in terms of a factor. In this section, the presented findings aim to answer RQ3 by determining a threshold level that could be used to potentially predict integration debt levels.

There may be different ways of creating thresholds for integration debt. This study's approach used historical data from the CI-system by grouping the data in a meaningful way (with specific intervals) and calculating the factor's ratio of that group against merge conflict. The data set used to evaluate the threshold is from the same project as the one used for measuring the factors of integration debt. Although the two data sets are from two different time spans which were selected; the first data set consisted of data from six months of development and the other was data from the latest two months of development.

Due to a limited time span of this study, it was only possible to investigate a threshold of one factor, which is the lines of code. LOC was chosen as it is one of the most basic factors and also one of the strongest factors (from the data analysis). However, it is important to keep in mind that the suggested threshold is project-specific and may be different for another project.

5.6.1 Threshold

Lines of code is a central factor to interaction debt, as indicated through the interviews, the questionnaire, and the repository mining. Beyond that, it became apparent that an increase in the size of LOC could increase the risks of merge conflicts, which adds up to integration debt. Thus, potentially dangerous levels of debt (where there is a higher risk of integration problems) were examined to determine if it is possible to create an accurate threshold. This simply means to know when the debt becomes critical in regards to lines of code. For this purpose, the data from the repository needed to be grouped in a meaningful way to allow for the determination of critical debt levels. Therefore, the LOC data from the CI-system over six months of development were grouped in 1000 LOC intervals and through the use of a script (see appendix A.4), all commits were classified in their respective group (LOC grouping) in a way to get a total count of commits belonged to each group. This is further explained in the subsection below.

After putting all the LOC data together, it became clear that after 10000 LOC, the data became vastly spread out. Moreover, commits that have between 0-10000 lines of code made up 98 percent of all the data. Therefore, only the data within the range of 0 - 10000 LOC is considered. Additionally, to get a better understanding of the data, LOC added and removed were investigated separately in the two subsections below.

5.6.1.1 LOC added

In this section, only the *added lines of code* in commits were included. This means the removed LOC in commits were not considered in this part. The data for LOC added was initially separated in groups with fixed intervals (1000 LOC). The sum of successful and merge conflict commits makes the total number of commits of that LOC group. From this data, the ratio of merge conflicts over the total count of commits for each group could be calculated. the ratio here can give somewhat of a likelihood of merge conflict for that group of LOC. For example, commits that have 0-1000 lines of code, have a 9,3 percent likelihood of causing merge conflicts. The ratio of each LOC group is plotted in Figure 5.38. These findings can assist in predicting future levels of merge conflicts within a commit.

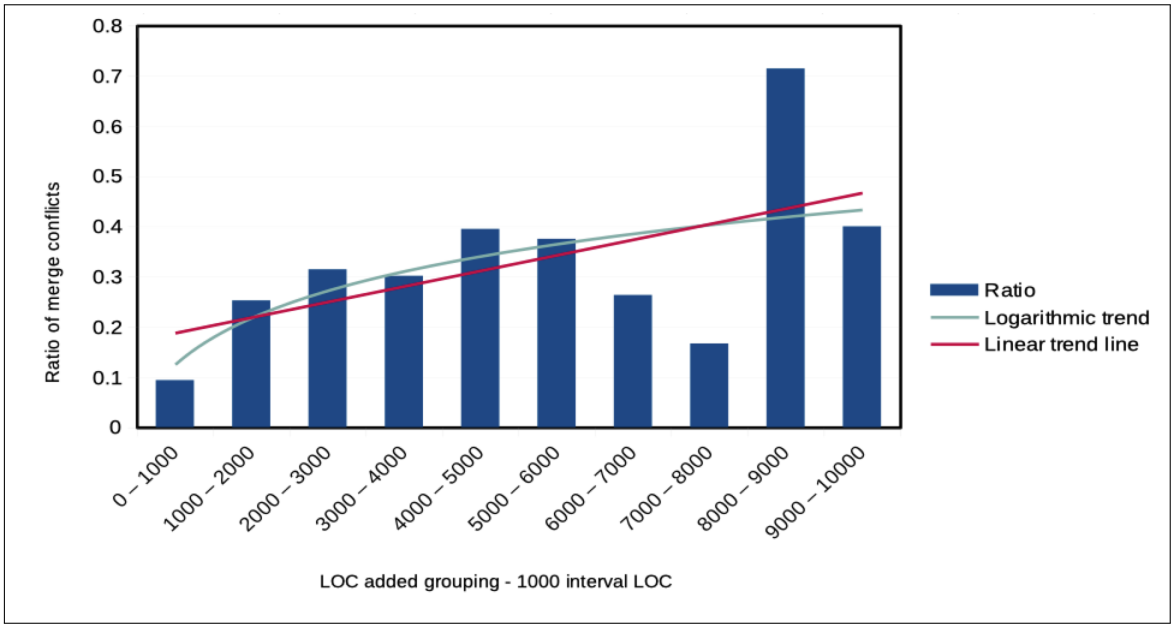


Figure 5.27: Grouped LOC and the ratio of merge conflicts against total count of successful commits for each group

Figure 5.38 seems to indicate a gradual increase in the trend of merge conflicts as the number of LOC increases. This can especially be seen after the first group of LOC, i.e. commits that have between 1000-2000 LOC. In this group, the ratio of

merge conflicts increases by more than 2.5 times. Also, the sudden increase of merge conflict ratio in the 8000-9000 LOC group could indicate a critical threshold level.

After a closer look at the data, it became obvious that judging the threshold level based on such a small sample size in these groups is not ideal. Thus making the results on the last groups not entirely accurate as the sample size is small. On the other hand, the majority of the data seems to be populated around the first three groups of LOC (0-3000). As a matter of fact these three groups comprise 99% of the data. Therefore another idea was to further investigate LOC added, by taking the more densely populated data that is between 0-3000 LOC and regrouping it in smaller intervals (of 100 LOC). This was done to show a more in-depth view of the impact of LOC on merge conflict. The findings are shown in the bar chart 5.28. The final findings here are also discussed later on in the discussion. There does seem to be an increase in merge conflicts once it reaches 1000 LOC. As mentioned previously, this is made especially clear in Figure 5.38. The threshold chosen for further investigation was thus 1000 LOC added.

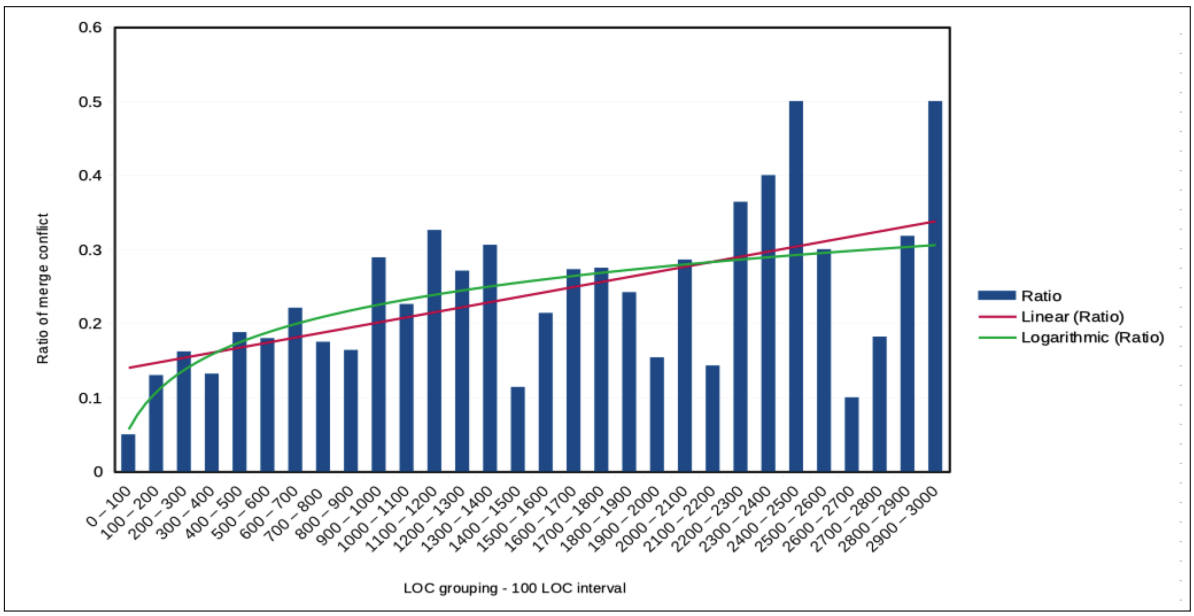


Figure 5.28: Grouped LOC and the ratio of merge conflicts against total number of successful commits - 100 LOC interval

5.6.1.2 LOC removed

This section is similar to the previous one. The only difference is that the considered data is *removed LOC* in commits. This means whichever commit that had at least 1 LOC removed over the designated 6 months of development, is used to calculate the impact of LOC removed against causing merge conflicts. Commits were counted with respect to their removed LOC and grouped in 1000 LOC intervals. This data was also plotted in Figure 5.39.

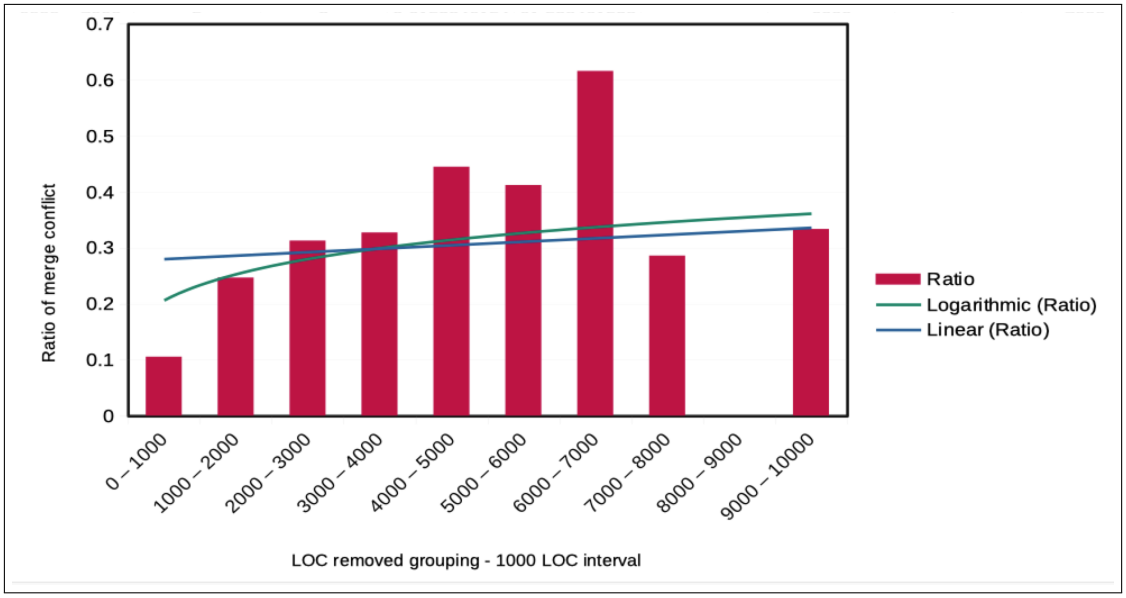


Figure 5.29: Grouped LOC and the ratio of merge conflicts against total count of successful commits for each group

By looking at the chart 5.39 above, a gradual increase within the first groups of LOC removed is visible, especially from the first to second group where the merge conflict ratio increases by almost 2.5 times. However compared to LOC added shown in chart 5.38, the steepness of the trend line for LOC removed is relatively lower. This indicates less significance of LOC removed on merge conflict, compared to LOC added. Additionally, similar to the previous section, the majority of the sample here is within the first three groups (0-3000 LOC). The data within this range contains more than 99% of the commits in LOC removed. The commits were grouped in 100 LOC intervals as plotted in the bar chart 5.30. The results are very similar for LOC removed as they were for LOC added. The same jump in merge conflicts once LOC reaches 1000 is made clear, especially in Figure 5.39. Thus, the threshold of 1000 LOC removed was chosen for further investigation.

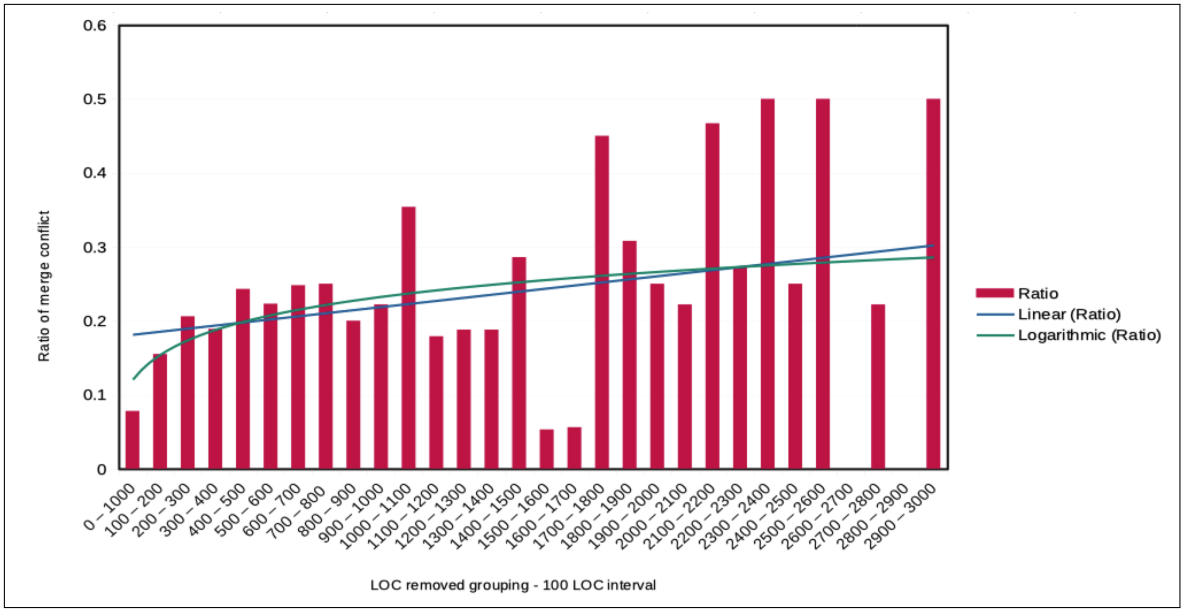


Figure 5.30: Grouped LOC and the ratio of merge conflicts against total number of successful commits - 100 LOC interval

5.7 Evaluation of LOC threshold using a second data set

In the previous section, thresholds of integration debt in regards to LOC within commits were chosen for further investigation. The data used there was from 6 months of development. To validate the suggested threshold levels, one approach is to use a different data set (with repository data from another time-period) and by grouping the data in a similar way to the previous section (5.6). The idea is to see whether the recommended threshold levels are applicable to another data set or in other words if the threshold is accurate or not. Thus this section tests the usefulness of the findings for thresholds.

The new data used was taken from the two most recent months of development. The grouping of data was done similarly to how threshold data were organized, i.e. firstly LOC added and removed are separately analyzed, as before. Secondly, the commits are again grouped in 1000 LOC intervals and depending on how much LOC a commit had, they were organized in their appropriate groups. Lastly, since the data is more concentrated in less than 3000 LOC, the data was also again grouped in smaller intervals of LOC for data between 0-3000 LOC. The findings regarding the total count of successful commits and the number of merge conflicts allow the possibility of calculating the ratio of merge conflicts for each factor. This is done by dividing the count of merge conflicts by the count of total commits for each group. This ratio gives a proportional value or a likelihood of merge conflicts regarding lines of code.

5.7.1 Threshold evaluation of added LOC using second data set

In this subsection, the results for LOC added are considered. The data used is from the new data set (two months of development). Of the total commits in the data set, more than 98% of them are below 10000 LOC added. Therefore only the commits that are between 0 - 10000 LOC added are taken into account. The sum of successful and merge conflict commits makes the total number of commits of that LOC group. From this data, the ratio of merge conflict over the total count of commits for each group could be calculated. The results from the table are plotted on the bar chart 5.31 below.

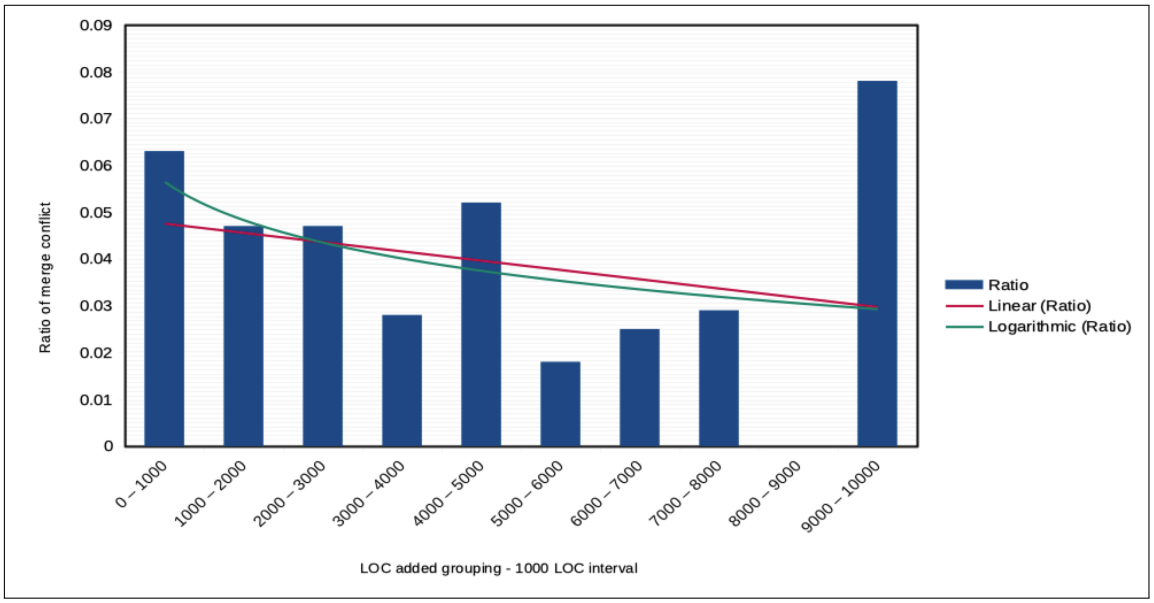


Figure 5.31: Grouped LOC and the ratio of merge conflicts against total number of successful commits - 1000 LOC interval

As the raw data suggests, the majority of commits belong to groups below 3000 LOC. This could be further investigated by breaking down the commits below 3000 in smaller groups. Thus commits from 0-3000 for LOC added were grouped in smaller groups (100 LOC intervals). The raw results for this further break down of findings are shown in appendix 5.31, although their bar chart 5.32 is displayed below. The same pattern in the first data set cannot be found in the second data set and thus the threshold is not accurate for LOC added.

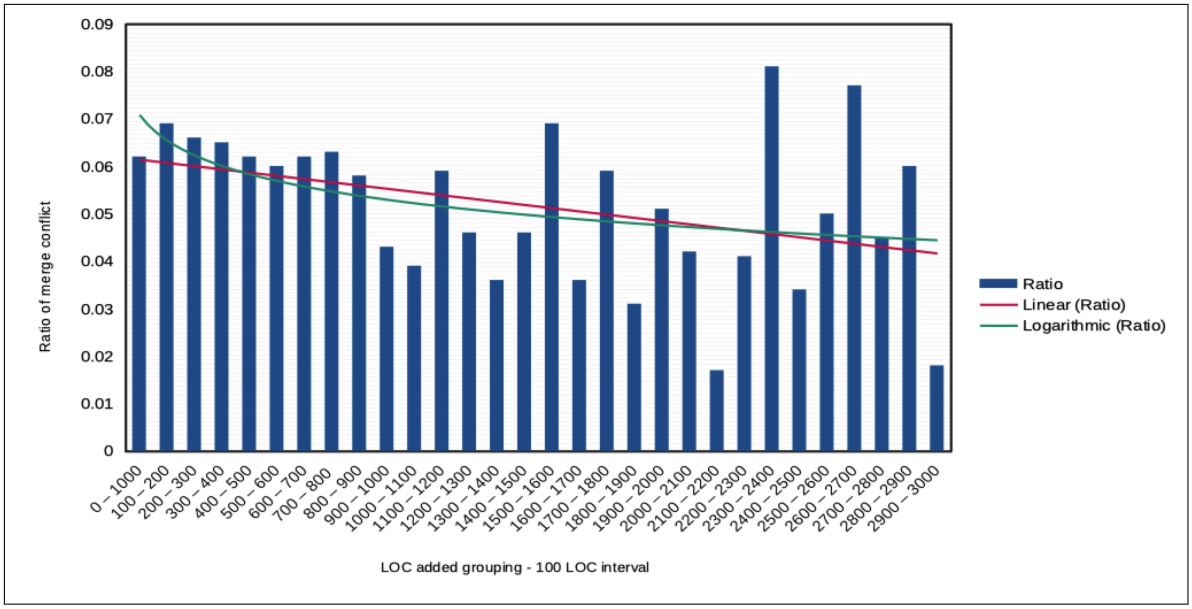


Figure 5.32: Grouped LOC and the ratio of merge conflicts against total number of successful commits - 100 LOC interval

5.7.2 Threshold evaluation of removed LOC using second data set

Similarly to how LOC added were grouped and analyzed, the same procedure is done here but instead on data regarding LOC removed. In this way, the impact of LOC added and remove can be individually seen. These results are plotted on the bar chart 5.33.

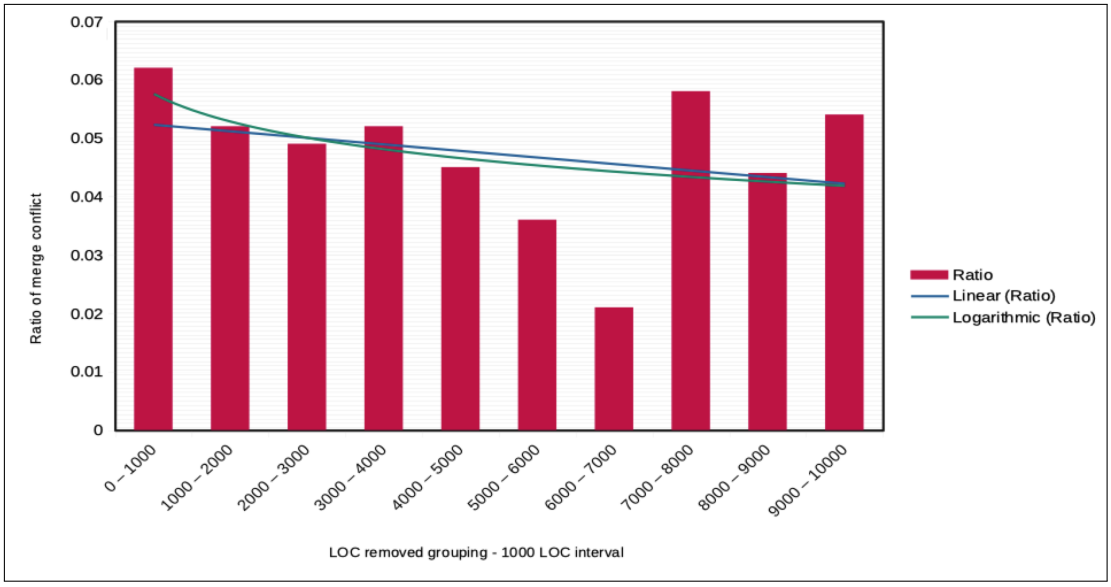


Figure 5.33: Grouped LOC and the ratio of merge conflicts against total number of successful commits - 1000 LOC interval

The next step was to further investigate the impact of LOC on merge conflict on commits below 3000 lines of code since data is largely populated between 0-3000 LOC range. Thus, data were grouped in 100 interval grouping. The raw results are plotted on bar chart 5.34. Similarly, as for LOC added, the pattern from the first data set is not visible in the second data set. This means that the threshold for LOC removed is not accurate either. These results are discussed in greater detail in the threshold sub-section in the discussion.

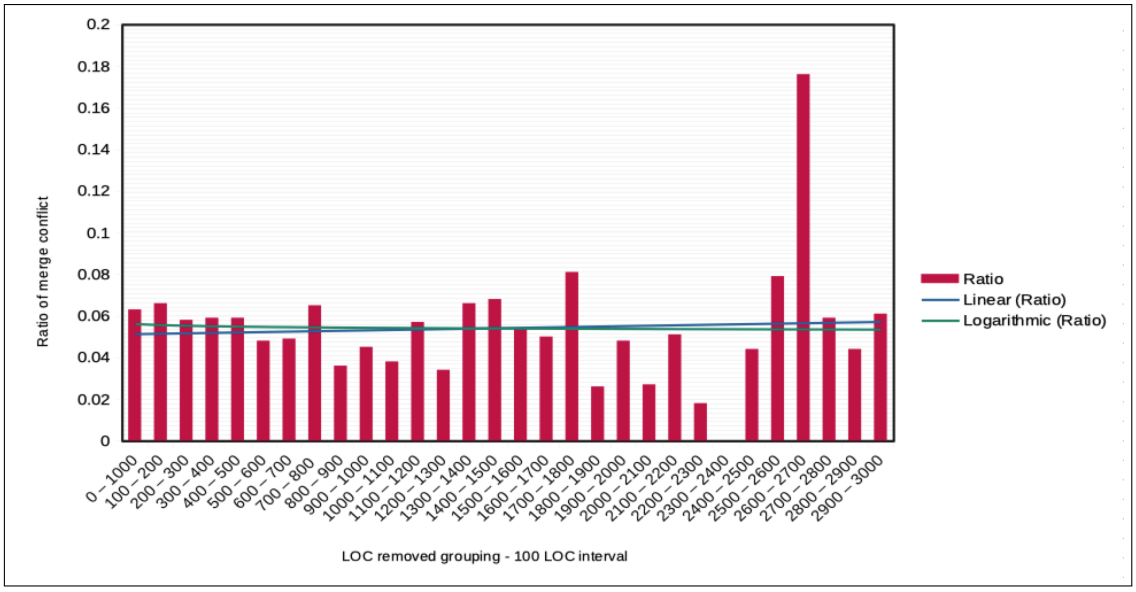


Figure 5.34: Grouped LOC and the ratio of merge conflicts against total number of successful commits

5.8 Summary of results

For the ease of readers, this section is dedicated to giving a summary of major findings in regards to the previously designed research questions. The findings aim to provide information to be able to answer the RQs.

5.8.1 Findings on RQ1

Research question 1 which is the basis of this study, is answered after a proper definition of the term integration debt is established and more importantly, its affecting factors are identified. In order to do that, different data gathering methods were used (e.g. interviews, repository mining). Figure 5.35 provides insights from the interview participants regarding potential factors of integration debt while Figure 5.36 gives an overview of the participants' answers in the multiple choice questions of the questionnaire. Lastly, table 5.3 gives a brief summary of the descriptive statistical analysis when measuring the correlation of factors within integration debt.

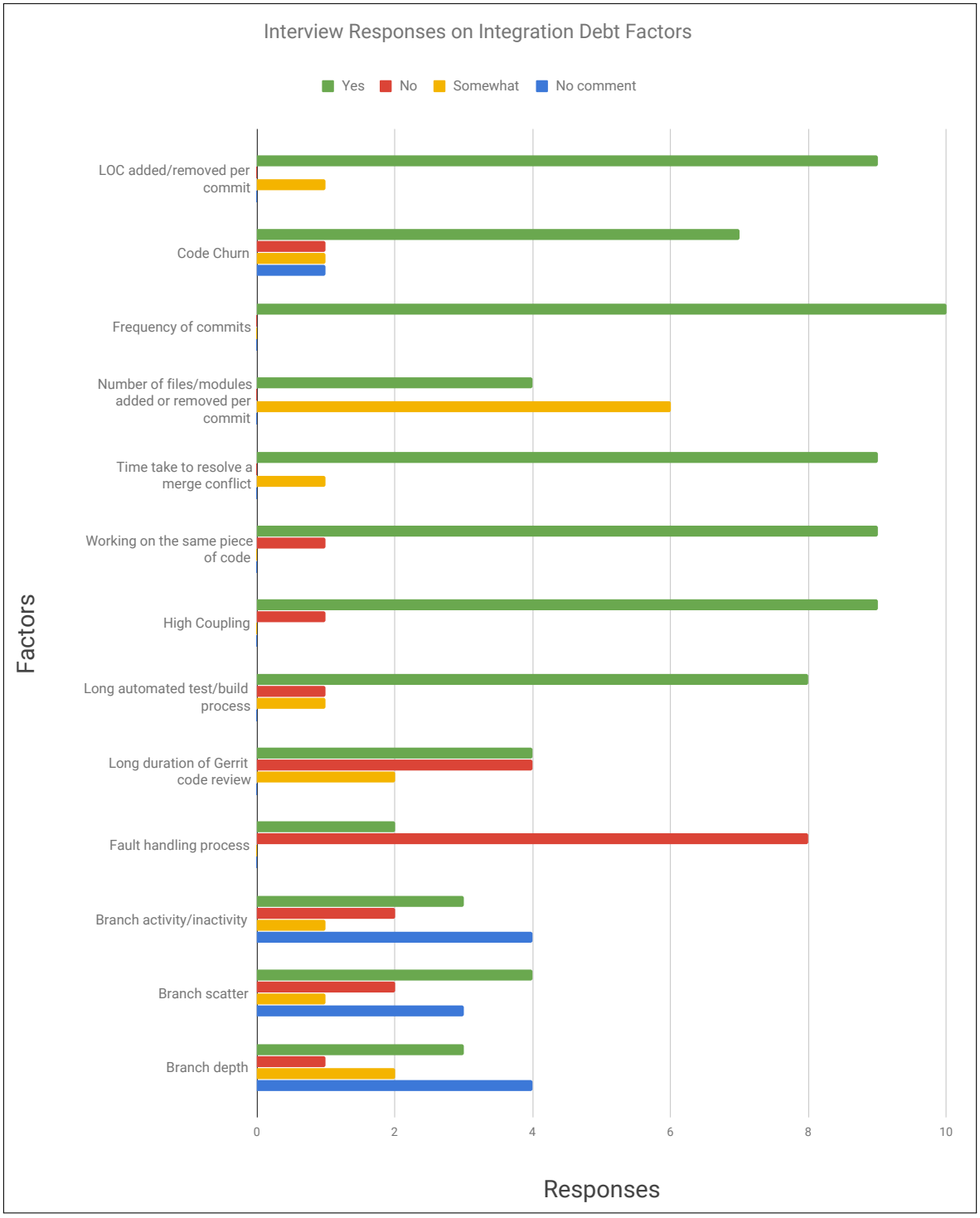
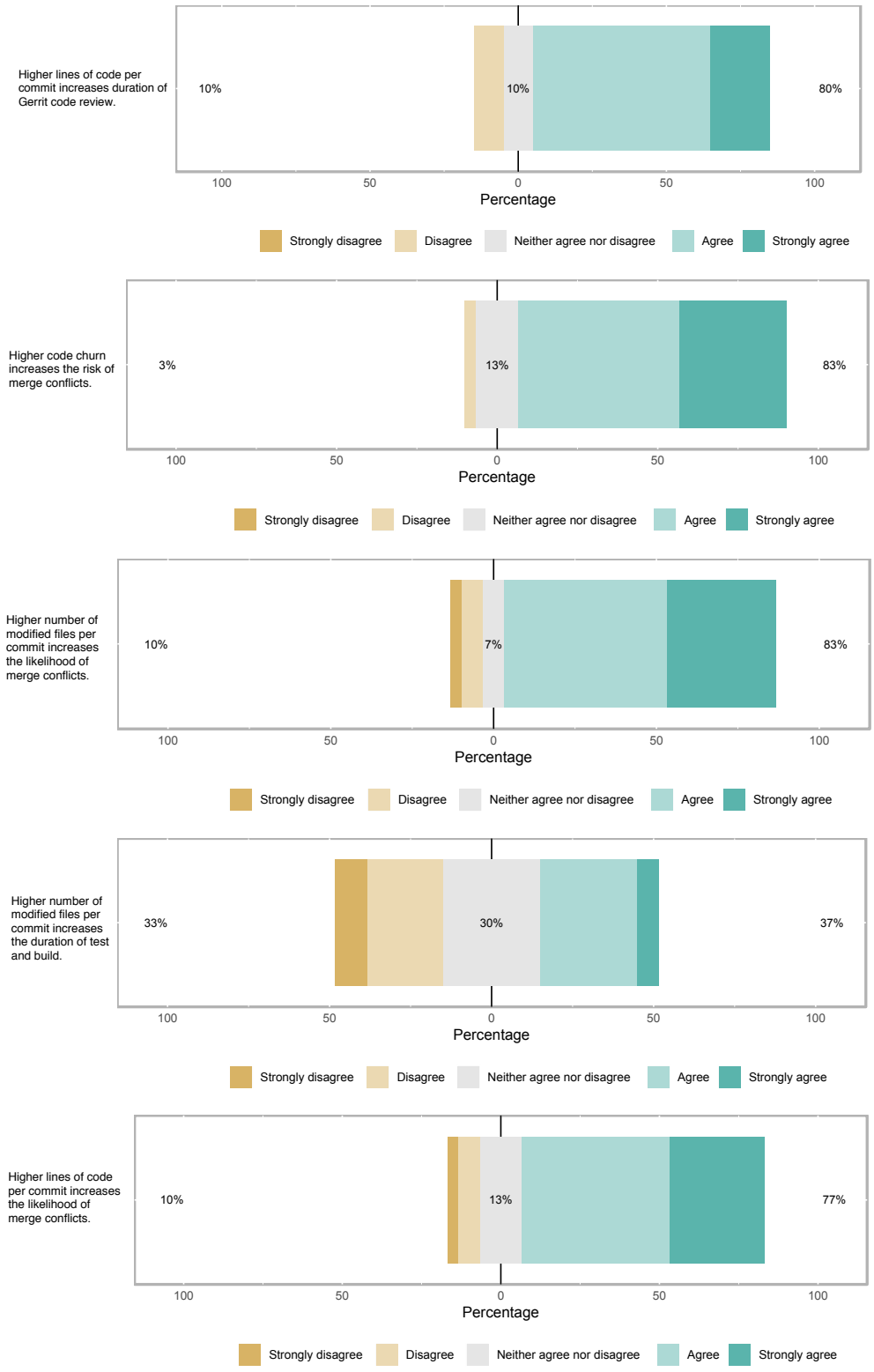


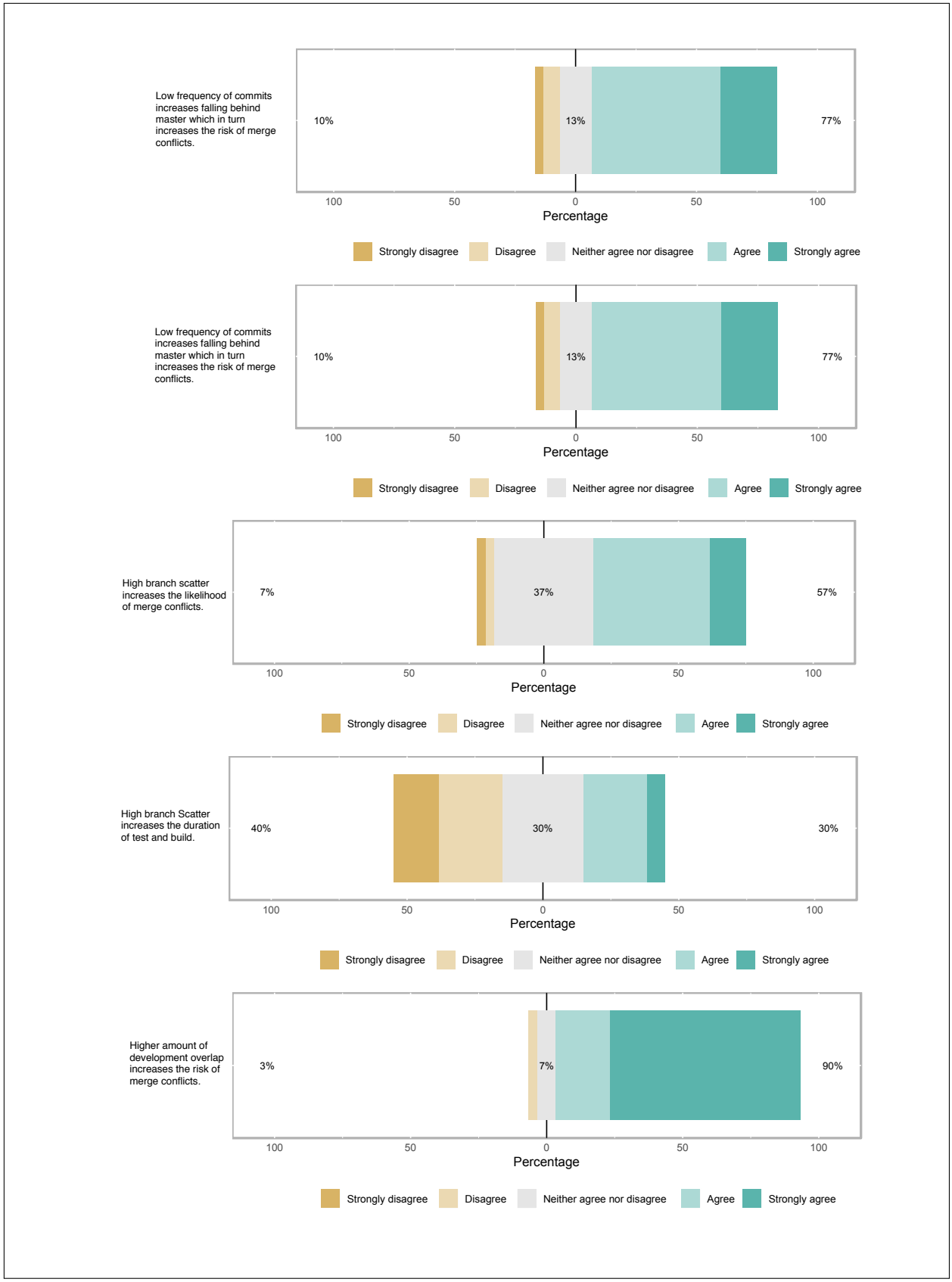
Figure 5.35: Interview responses about whether or not the factors gathered are actually integration debt factors

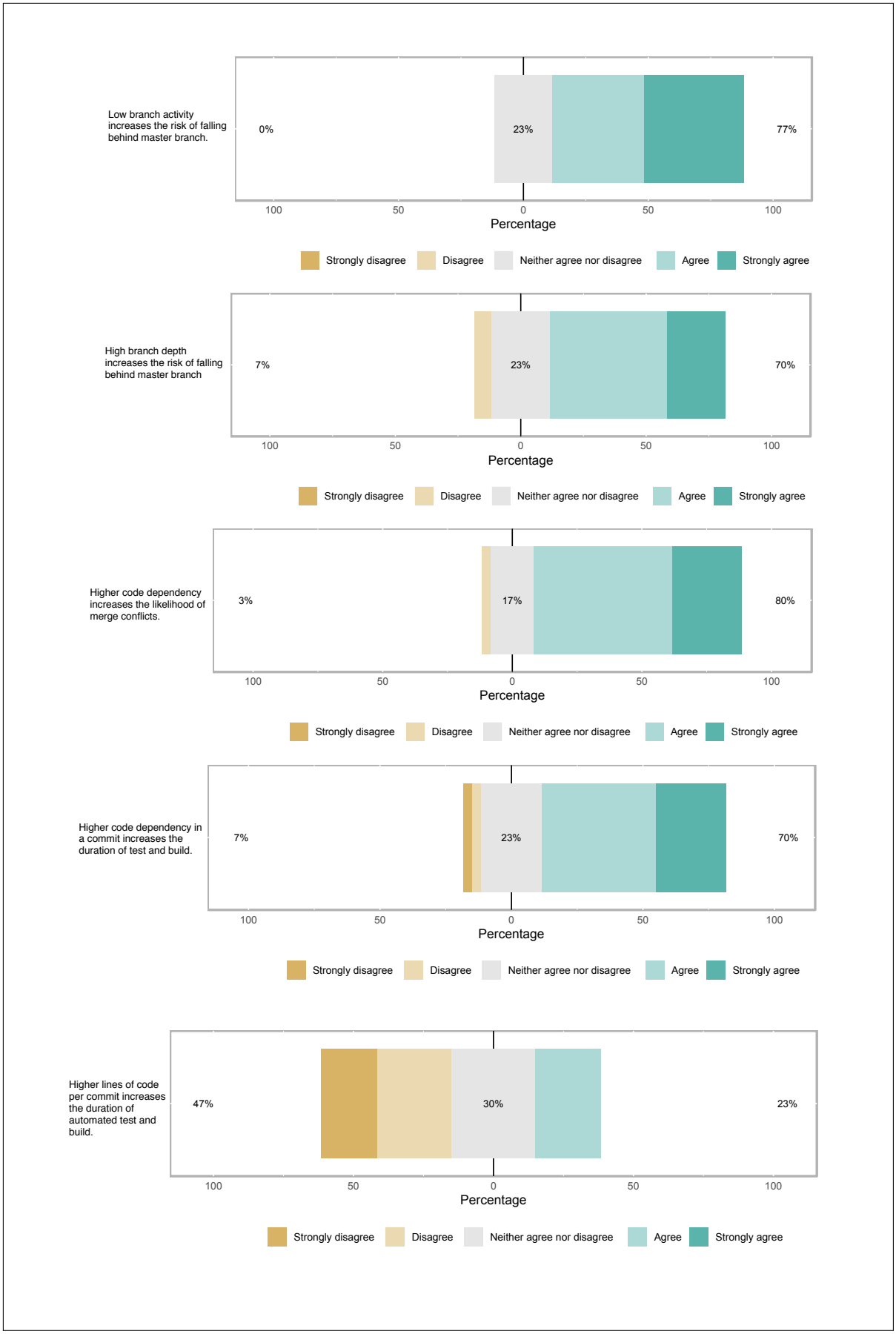
Factor	Normal Distribution (Shapairo Wilk's W)	Spearman rank correlation (rho)	Pearson correlation coefficient (r)	T-test's p-value	Basic interpretation
Lines of code versus merge conflict	LOC added: 0.00000001719 LOC removed: 0.00000002062 Both Not normally distributed	LOC added: 0.5579474 LOC removed: 0.5327245	-	LOC added: 0.00461 LOC removed: 0.00736	Moderately strong correlation, statistically significant result
Lines of code versus duration of automated test & build	LOC added: 0.00000001719 LOC removed: 0.00000002062 Both not normally distributed	LOC added: 0.4295652 LOC removed: 0.2591304	-	LOC added: 0.03723 LOC removed: 0.2205	Moderate correlation and statistically significant result for LOC added, weak correlation for LOC removed
Code Churn (total of LOC added and removed)	LOC added: 0.00000001719 LOC removed: 0.00000002062 Both not normally distributed	0.5596869	-	0.00457	Moderately strong correlation, statistically significant result
Number of files versus merge conflict	0.4494 Normally distributed	-	0.5643074	0.004073	Moderately strong correlation, statistically significant result
Number of files versus duration of automated test & build	0.1349 Normally distributed	-	0.3839306	0.0640	Moderately weak correlation
Frequency of	0.1447	0.6800936			Strong correlation,

commit versus merge conflict	Normally distributed		-	0.0002559	statistically significant result
Branch scatter against merge conflict	0.0317 Not normally distributed	0.2552729	-	0.2286	Very weak correlation
Branch scatter versus duration of automated test & build	0.0317 Normally distributed	0.1652174	-	0.4387	Very weak correlation

Table 5.3: Summary of statistical analysis on some of the factors


Higher lines of code per commit increases duration of Gerrit code review.
10%
10%
80%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher code churn increases the risk of merge conflicts.
3%
13%
83%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher number of modified files per commit increases the likelihood of merge conflicts.
10%
7%
83%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher number of modified files per commit increases the duration of test and build.
33%
30%
37%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher lines of code per commit increases the likelihood of merge conflicts.
10%
13%
77%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree


Low frequency of commits increases falling behind master which in turn increases the risk of merge conflicts.
10%
13%
77%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Low frequency of commits increases falling behind master which in turn increases the risk of merge conflicts.
10%
13%
77%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
High branch scatter increases the likelihood of merge conflicts.
7%
37%
57%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
High branch Scatter increases the duration of test and build.
40%
30%
30%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher amount of development overlap increases the risk of merge conflicts.
3%
7%
90%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree


Low branch activity increases the risk of falling behind master branch.
0%
23%
77%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
High branch depth increases the risk of falling behind master branch
7%
23%
70%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher code dependency increases the likelihood of merge conflicts.
3%
17%
80%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher code dependency in a commit increases the duration of test and build.
7%
23%
70%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree
Higher lines of code per commit increases the duration of automated test and build.
47%
30%
23%
100
50
0
50
100
Percentage
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

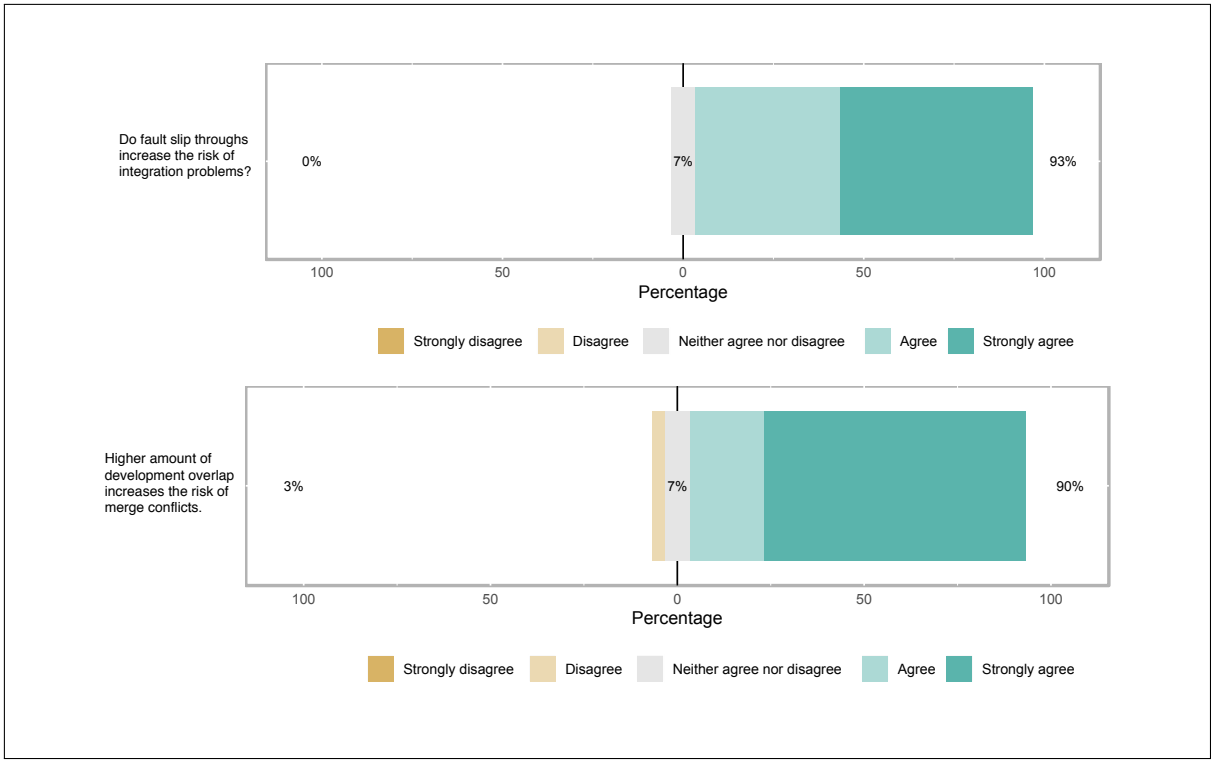


Figure 5.36: Summary of results from questionnaire - multiple choice questions

5.8.2 Findings on RQ2

In this sub-section, the findings that lead to answering RQ2 are provided. RQ2 is built upon the first research question which is after measuring and potentially mapping the affecting factors of integration debt. For this purpose, assessment of the overall findings from used methods was required to get an understanding of which factors are proven to affect integration debt and which ones do not. The findings are from methods such as; literature review, open-ended interviews with CI-managers, interviews with developers, repository mining and a questionnaire. The summary of the findings of each investigated factor is mentioned in the three tables below. Each table is distinct in a way that, one displays the approved factors (table 5.4), another the inconclusive factors (table 5.5) and the other rejected factors (table 5.6). The reason for separating factors was to clearly show which ones were approved as integration debt factors as well as the inconclusive ones which due to lack of sufficient evidence were unable to be approved or disproved as a factor. Similarly, the factors that were enough evidence to reject them as an integration debt factor. Additionally, in the tables, the methods used for evaluating a factor are shown in the middle column. Results for the factors that have been measured using more methods, tend to be more reliable. In the right-most column of the tables below, a commentary section exists where describes the reasoning behind why a factor is evaluated as such.

Approved integration debt factors		
Factor	**Adopted methods of evaluation**	**Commentary**
Lines of code on causing merge conflict	Literature review, Interview, repository mining and questionnaire	Strong conviction in LOC being a factor in integration debt. The findings showed that as LOC increases within a commit, the likelihood of causing merge conflict also increases. Findings also showed that removed LOC in commits have lower impact than added LOC.
Number of modified files on causing merge conflict	Literature review, Interview, repository mining and questionnaire	Reasonably strong statistical correlation between NOF and occurrence of merge conflicts. Interview and questionnaire findings suggest the same that NOF is a factor in integration debt.
Code Churn on causing merge conflict	Literature review, Interview, repository mining and questionnaire	Findings from all adopted methods, indicate that a developer can increase the risk of merge conflict by frequently modifying a file (in terms on LOC) in short time span. Thus code churn is a factor in integration debt.
Frequency of commits	Literature review, Interview, repository mining and questionnaire	Strong results from statistical analysis of the mined data together with findings from interview and questionnaire, suggest the frequency of commits have a direct correlation with cause of merge conflicts. Low frequency of commits increases falling behind master branch which in turn increases the risk of merge conflicts.
Code dependency	Literature review, Interview and questionnaire	Despite lack of statistical analysis, other findings seem to indicate the involvement of code dependency on integration debt. Developers and CI managers at the company seemed to recognise dependency as a factor that can not only increase the occurrence of cause merge conflicts but could also affect the duration of test & build.
Development overlap	Literature review, Interview and questionnaire	Strong findings from developers and CI managers pointed to impact of development overlap on integration debt. Majority of the involved participants recognised that working in parallel on a same piece of code together with lack of communication can lead to merge conflicts.
CI-system's factors	Discussion with CI managers, Interview and questionnaire	Developers and CI managers at the company seemed to strongly consider them as integration debt factors. Despite lack of statistical analysis, involvement of CI-system's factors is vivid in prolonging duration of integration. However, their level of impact on integration debt remains uninvestigated.

Table 5.4: Finalised factors of integration debt

Inconclusive integration debt factors		
Factor	**Adopted methods of evaluation**	**Commentary**
Branch scatter on causing merge conflict	Literature review, Interview, repository mining and questionnaire	Branching factors were difficult to evaluate since using branches were not a common approach used by developers in the company. The result from all methods were not strong enough to indicate a relationship to integration debt. Although there was not solid findings to disregard as a factor, therefore it remains indecisive and a potential factor of integration debt.
Branch activity	Literature review, Interview, repository mining and questionnaire	Findings from questionnaire showed that developers do think of branch activity as a factor in integration debt. Although due to lack of statistical analysis and inconclusive results from interview, there not enough proof to consider it as a factor. This was accompanied by inconsistency of using branches during development.
Branch depth	Literature review, Interview and questionnaire	All findings were considered as inconclusive, mainly because the developers at the company did not use branches and the ones who did, appeared to not use branches on top of each other. Thus, branch depth was neither accepted nor rejected as a factor and remains uncertain as a potential factor of integration debt.

Table 5.5: Finalised inconclusive factors of integration debt

Rejected integration debt factors		
Factor	**Adopted methods of evaluation**	**Commentary**
Lines of code on duration of automated test & build	Literature review, Interview, repository mining and questionnaire	Findings from statistical analysis on the mined repository data showed that there is very little (almost non-existing) relationship between LOC and duration of test & build. Questionnaire result also pointed to the same realization.
Number of modified files on duration of automated test & build	Literature review, Interview, repository mining and questionnaire	Interview and questionnaire results were inconclusive but results from repository mining showed very little correlation between NOF and duration of test & build in the project. Thus, NOF does not appear to have an impact on the time taken in the test and build stage.
Fault handling process (debugging)	Discussion with CI managers and Interview	Almost all participants in interview rejected debugging as a factor in integration debt. Majority believed that fault handling is part of a developer's daily tasks. Due to strong findings in interview, fault handling was not further investigated and was disregarded as a factor.
Branch scatter on duration of automated test & build	Literature review, Interview, repository mining and questionnaire	A very low statistical correlation value together with questionnaire and interview result led to disregarding branch scatter as a factor on duration of test and build.
Duration of Gerrit code review	Discussion with CI managers, Interview and questionnaire	Even though that some participants did think that more LOC in a commit often leads to longer duration of code review, they did not see it as a factor in integration debt.

Table 5.6: Rejected factors of integration debt

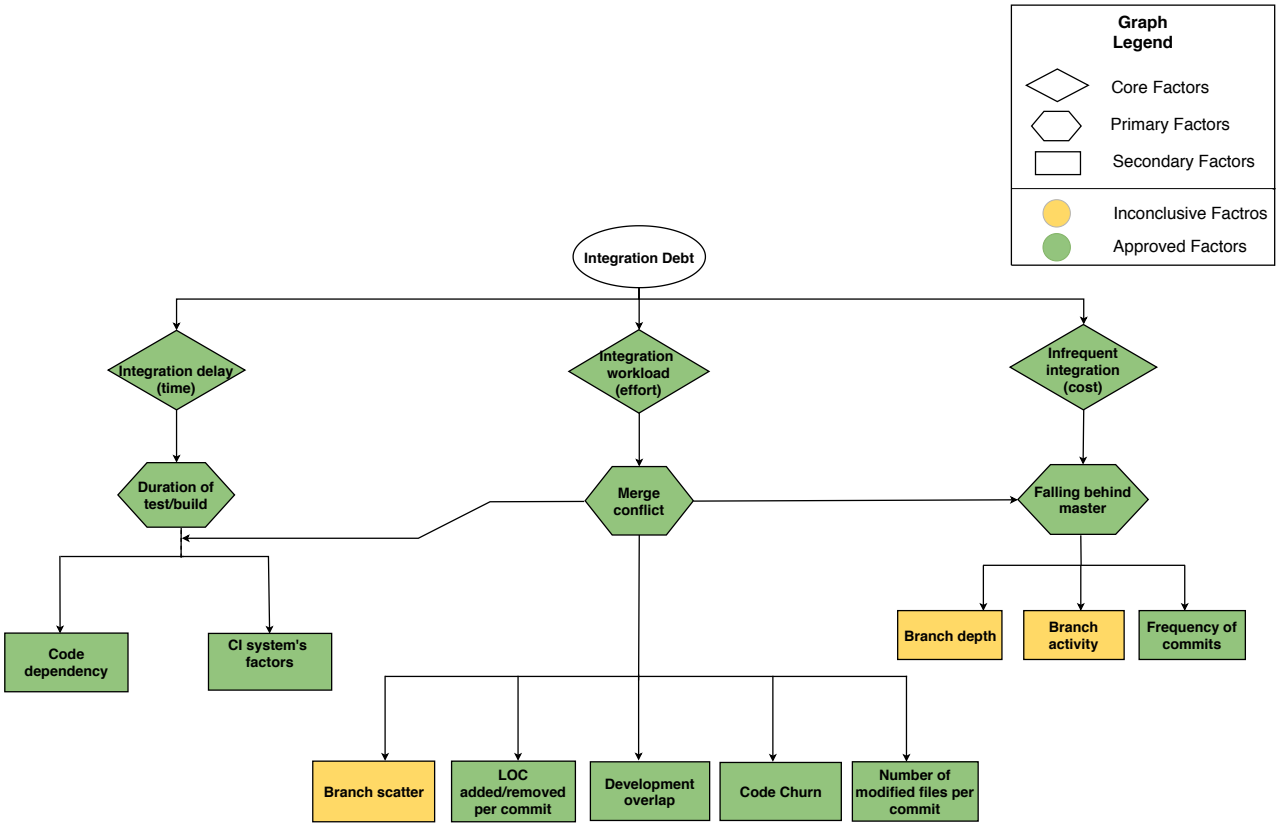


Figure 5.37: Finalized mapping of factors

This section also provides the finalized mapping of integration debt shown in figure 5.37. This figure is based on its initial version that was only based on findings from interviews and literature review shown earlier in the results chapter (figure 5.4). The finalized version of the mapping took into account findings from all methods of data gathering, and after a detailed analysis it was refined and the relationships were re-adjusted. The mapping only consists of the approved factors (5.4) shown in green color and inconclusive factors (5.5) shown in yellow color. The rejected factors (5.6) were left out since there was enough evidence to reject them as integration debt factors. The reason for including inconclusive factors in the mapping figure is due to insufficient findings to reject or approve those factors. Thus the inconclusive factors are still considered as potential factors that require further investigation. For further interpretation and analysis of the mapping figure, refer to section 6.1.3.

5.8.3 Findings on RQ3

The focus here is to display the findings in answering RQ3, which is determining if a threshold of an integration debt factor can be found in order to predict the debt levels. To do this, the factor of lines of code (LOC) was chosen for an investigation of if an accurate threshold can be found. The results are shown in section 5.6. In a nutshell, after grouping the commits' lines of code in 1000 LOC intervals and

comparing the number of merge conflicts against the total number of commits for each group, the threshold level was identified. The findings are displayed below, for added LOC in figure 5.38 and removed LOC in 5.39. The raw data was mainly populated in the first three groups (0-3000 LOC), making the results more reliable compared to the right-most groups. The conclusion regarding the threshold level is the commits that above 1000 LOC as the ratio of merge conflicts increases by more than 2.5 times in both LOC added and removed. Although it became clear that more investigation was needed, perhaps by analyzing more data to get an average of all findings to determine a more reliable threshold, or by using other methods. This is because the threshold did not hold when evaluating it using the second data set, suggesting that the answer to RQ3 is inconclusive at best, and no at worst. The discussion regarding the threshold is continued in the next chapter 6.1.4.

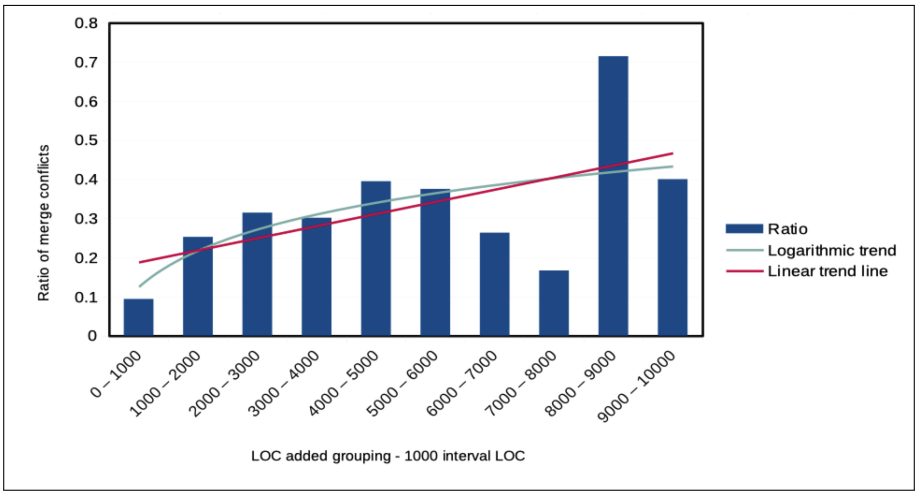


Figure 5.38: Grouped LOC added and the ratio of merge conflicts against total count of successful commits for each group

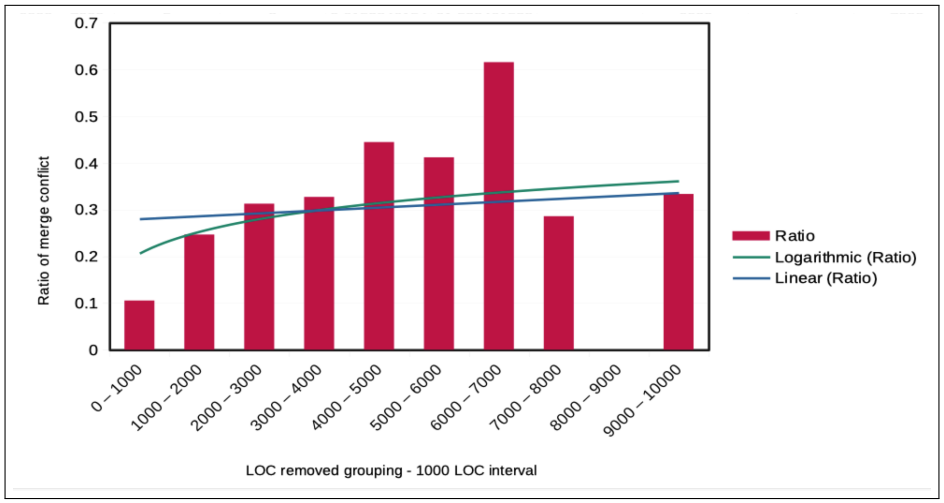


Figure 5.39: Grouped LOC removed and the ratio of merge conflicts against total count of successful commits for each group

6

Discussion

This chapter is dedicated to interpreting the results and their impact on the research community as well as organizations that adopt CI in their software development and are looking for ways to optimize code integration. Additionally, integration debt and its involved factors, approaches used to investigate integration debt, the applicability of findings, difficulties encountered and threats to validity of findings are discussed.

6.1 Interpretation of results

In this section of the study, the findings that allow answering the research questions are analyzed and discussed in detail. In order to do that, integration debt is appropriately defined, the findings on factors are analyzed, factor relationships to integration debt are accordingly mapped and lastly an integration debt threshold in regards to lines of code in commits is determined.

6.1.1 The integration debt term and its definition

Initially, the involved company in this study had expressed that branches working in isolation for too long appear to add extra costs and effort to the company. They referred to it as integration debt and the company was interested to comprehend what affects branches (or developers) to fall behind the master branch and develop in isolation for too long. They were interested to monitor the integration debt levels across their large-scale project and be able to take actions to mitigate the debt. To do that, the first step was to carry out a literature review on the domain area. Surprisingly, only one study [22] was found that had mentioned the term integration debt. Although that study only refers to delays in code integration, as integration debt and does not discuss further on what the term actually means and how it occurs. That is the purpose of this study, which aims to define integration debt, identify its causing factors and suggest how it could possibly be measured. Defining integration debt and its factors was done to answer RQ1 and it was done through

various data gathering methods as described earlier in the results section. However, a summary of the major findings to answer RQ1 is shown in section 5.8.1.

Since integration debt was not a well-established notion, the metaphor of technical debt was used and applied to continuous integration in order to provide a preliminary definition of integration debt. Technical debt means implementing a suboptimal solution now (taking on debt) which will require interest to be paid (in terms of negative consequences of the suboptimal solution) on it over time. Applying this to CI development would mean having unintegrated code (having debt) which requires interest to be paid (in the form of an increasing risk of integration problems) on it over time. This is the fundamental idea and definition of integration debt in this study. However, there are also other potential definitions or developments of this fundamental definition which could also be used.

One alternative, or perhaps complementary, definition would be that integration debt is a measure of how far away (in terms of time, effort, cost) you are from integrating your unintegrated code. It is similar to the previously explained fundamental definition in the sense that the unintegrated code needing to be integrated is the underlying problem and there are obstacles in the way. For this definition, the obstacles are described as the distance to integration whereas, for the fundamental definition, the obstacle is the (increasing) interest on the debt. This definition can be derived by looking at the integration debt factor mapping 5.37, and seeing that the factors all lead to the core factors which represent time, effort and cost. The levels, or values, of the secondary factors, will affect the primary factors which in turn affect the core factors as integration happens. As such a greater level of integration debt (across the secondary factors) would lead to a greater "cost" for the core factors and would thus mean you are further away from integration since you need to pay a higher price in terms of time, effort or cost.

Another version of the definition would be that integration debt is simply a measure of the risk of integration problems at a certain point in time. This is essentially just a simplification of the previously mentioned definition but it does provide a quick and easy to understand explanation of the problem that the integration debt term describes. In the next sub-section, the results of the investigated factors are discussed.

6.1.2 Factors

This section of the discussion aims to further interpret and discuss the results of RQ1, i.e. to identify the factors involved in integration debt. These potential factors are aspects of development in the context of continuous integration. Understanding the causing factors ultimately allowed a better comprehension of the term *integration debt* itself. To recognize these factors in a reliable manner, a set of methods were adopted that incrementally added to our knowledge and understanding of each in-

tegration debt factor. These methods are grouped in iterations which are explicitly described in the methodology section. Note that not all factors used all the methods of evaluation, for instance *branch depth* was not statistically measured, due to a lack of available data.

The different methods of data gathering led to some contradictory findings as in some cases the results from one method disagreed with results from another method. This was mostly between statistical results (quantitative) and interview/questionnaire responses (qualitative), For instance; when collecting data to measure branch scatter as a factor of integration debt, statistical findings indicated a relatively weak correlation in causing merge conflicts. Meanwhile, participants involved in the questionnaire seemed to believe branch scatter has an effect on causing merge conflicts. The difference between results can be caused by various reasons. One is the participants' (developers') level of knowledge or experience that affected their responses during the questionnaire. For instance, a developer may view a certain factor through their own limited and anecdotal experience while the picture painted by the statistical evidence might be entirely different. On the other hand, the statistical analysis can not, and does not, take into account many variables that may affect an integration debt factor which may be known by a developer. Neither source can be dismissed and both the quantitative and the qualitative results were weighed against each other on a case by case basis for each factor.

These contradictions in the findings caused some issues with analyzing the results from the data gathering, but by taking into account the results of all other data gathering methods, there was more evidence to examine the factors more reliably. Although it is important to note that in many cases the statistical data was deemed more accurate and trustworthy due to its objectivity, as such the findings there are often prioritized compared to the qualitative findings.

It is worth noting that most of the responses in the questionnaire were positive that the mentioned integration debt factors were indeed factors. There could be several explanations for this. To begin with, the set of integration debt factors gathered up until that point had been refined and one possibility is that they are indeed interesting factors to consider for integration debt and that the responses simply reflect that. Another explanation could be that the developers were presented with these factors by the researchers and did not think that they had enough of a reason to disagree with the researchers about these factors being potential integration debt factors. Also, the developers' understanding of the factors and the integration debt term may have led them to not want to prematurely dismiss a factor that they may have had some experience with, within the context of continuous integration.

6.1.2.1 Lines of code

Lines of code is the first factor that is being interpreted and discussed. This is in terms of both LOC added and removed for each commit. LOC was originally

identified from a report [19], where it claimed that an increase in the number of LOC could potentially increase occurrences of defects which could affect the duration of test and build, and other integration issues such as merge conflicts. From the interview, 9 out of 10 participants believed LOC added/removed per commit is a factor in integration debt. The majority believed that bigger commits have a higher risk of merge conflicts.

One interesting observation was that participants referred to the size of the commit as being the amount of LOC rather than any other potential metric such as the number of files. The fact that LOC was such a ubiquitously understood and used metric among the developers indicates that it is considered to be an important and useful metric. A potential issue with evaluating LOC may, therefore, be that it could be overrated by the developers as an integration debt factor. On the other hand, the familiarity the developers have with the metric of LOC may have allowed them to have a stronger understanding of LOC as a potential integration debt factor. This can be contrasted with other factors, such as branch scatter, which are less straightforward and which the developers were not as familiar with.

The data mining results for LOC were a moderately strong correlation with merge conflicts (rho = 0.557 for LOC added and rho = 0.532 for LOC removed). For the relationship between LOC and duration of test/build, the correlations were found to be a rho of 0.429 for LOC added and a rho of 0.259 for LOC removed. An interesting observation to make about this result is that LOC added has a very similar impact on merge conflicts as LOC removed but that there is a major difference in their impact on test/build duration where LOC removed has a much smaller impact than LOC added. A potential explanation for this is that as LOC is added there are also defects introduced, as noted by Nagappan & Ball [19]. These defects then cause errors during test and build, which need to be fixed, and more test cases may need to be run which increases the duration. When removing LOC, there is no new code (which could contain defects) being introduced and as such it does not increase the duration as much. Removing LOC may still cause code to break and may still cause errors to happen during test and build which could explain the weak correlation that does exist for LOC removed and duration of test/build.

From the questionnaire results, most participants (77%) agreed that LOC increases the risk of merge conflicts but only 23% agreed that it increases the duration of test and build. It would seem that both the developers and the statistics agree that LOC is a definite factor in merge conflicts but that it is less of a factor in the duration of test and build. As mentioned previously, LOC removed has an especially weak relationship with the duration of test and build.

6.1.2.2 Number of files

The modified number of files (NOF) was another potential factor of integration debt. Files refer to any type of code modules that are submitted to be merged to a

master branch. NOF was originally perceived from literature [19], where similar to LOC, was discovered as a factor of causing defects. This gave the idea of investigating whether an increase in the number of files could affect the likelihood of merge conflicts or perhaps even increase the duration of automated test & build.

From the interview, when developers were asked whether NOF is a factor in integration debt, 4 out of 10 believed NOF per commit has an impact on integration debt and the rest were indecisive. A few participants had a similar idea, that files removed or touched, will have an impact on integration debt while adding files does not really increase the debt levels. An interpretation of this statement could be that removing or modifying a file could affect the dependency (coupling) of the entire system and that could cause integration problems. Another significant comment from a participant was that the more files that are being merged to a repository, the more test cases are triggered which naturally makes the duration of test and build longer and the feedback loop longer.

The results from the data mining and analysis showed a moderately strong correlation ($r = 0.564$) between NOF and merge conflicts, and a weak correlation ($r = 0.165$) between NOF and duration of test and build. A potential reason for why this is the case, is that adding or modifying files does not necessarily increase the risk of errors, or the triggering of tests, during test and build since it does not matter that much if code is split into multiple files or not for the purposes of testing and building, it is still the same code. This can be contrasted with merge conflicts where a merge conflict may happen due to a file being changed, moved, removed or renamed.

Results from the questionnaire showed that most participants (83%) agreed that higher NOF increases the risk of merge conflicts but that it is not much of a factor in the duration of test and build, with only 37% of participants saying that it is a factor. Similarly, as for LOC, both the developers and the statistical analysis agree that NOF affects merge conflict risk but does not have a major impact on the duration of test and build.

6.1.2.3 Frequency of commits

In this study, frequency of commits is defined as how often commits have been pushed to the master branch within a time period, e.g. 10 commits in 5 days is 2 commits per day on average. The idea of the frequency of commits being involved as a factor in integration debt originated from reviewing literature [20]. The report stated that temporal proximity of commits could cause more problems and defects. Thus the assumption of whether committing more or less affects the occurrence of merge conflicts was put to the test.

From the interview results, 10 out of 10 participants believed that the frequency of commits is a factor in integration debt. All participants recognized that smaller intervals between commits (frequent small commits), could lower integration prob-

lems, Meanwhile a few argued that it could also affect the efficiency of the CI-system. Since more commits require more CI resources, committing too often may not be an ideal approach.

The data mining and statistical analysis revealed a fairly strong correlation ($r = 0.68$) between the frequency of commits and merge conflicts. In fact, this relationship is the strongest of all the investigated potential integration debt factor relationships. The reason for this may be that a higher frequency of commits also means a higher frequency of potential merge conflict commits.

It is important to note that branch activity and frequency of commits (two separate factors) have some overlap in their definitions. The result for the frequency of commits showed that more commits in a time period increase the risk of merge conflicts but it does not necessarily go against the assumption of low branch activity (i.e. long period without committing) increasing the risk of merge conflicts. This is simply because a developer can still fall behind the master branch by committing less often.

These results do perhaps suggest that it would be interesting in future research to further investigate the frequency of commits versus branch activity as defined in this study. Committing more does seem to increase integration problems, in this case, merge conflicts, as shown by the statistical results and as noted in literature [20]. However, this goes against the general principle of CI of committing often. The contradiction here might have arisen as a consequence of a lack of granularity in the terms in the sense that committing more often does not mean committing more since the developer still develops the same amount of code and the only difference is when they commit. Thus frequency of commits might be too simplistic and should perhaps also simultaneously consider the development rate in order to be able to distinguish between more commits being pushed due to more code being written, and more commits being pushed due to more frequent integration.

Lastly, from the questionnaire, participants seemed to agree on the impact of the frequency of commits on the risk of merge conflicts. 77% of the participants agreed, while 13% were indecisive and only 10% disagreed with the frequency of commits being a factor in merge conflicts. Once again, both the developers and the statistics agree with each other, in this case, that the frequency of commits definitely affects merge conflicts. It can therefore be considered a significant integration debt factor.

6.1.2.4 Branch Scatter

Branch scatter is defined as how spread out the developed is across branches. The assumption of branch scatter being involved in integration debt was initiated from reviewing literature [21]. In this study, the effect of branching on software quality is measured. One realization was that an increase in branch scatter was labeled as a potential sign of an increase in the number of defects in a repository. Therefore,

as it is displayed in factor mapping figure 5.4, branch scatter was measured against merge conflicts and duration of automated test and build.

The results from the interviews when participants were asked about the involvement of branch scatter on integration debt, were diverse. This was mainly because not everyone used branches in their development and they were, therefore, less familiar with branches and the concept of branch scatter. Thus the findings from the interviews remained rather vague as many developers did not have an opinion on branch scatter being a potential integration debt factor.

In the statistical analysis of the mined data, branch scatter was shown to have a weak relationship with both merge conflicts (rho = 0.255) and with the duration of test and build (rho = 0.165). This may partly be explained by the relative lack of branching workflows among the developers which meant that there were not many large and separate branches (they were mostly small personal branches) that were used for development. It may have resulted in the branch scatter having less impact on integration problems. Another observation to be made is that the effect branch scatter had on the duration of test and build was very small which would suggest that it does not introduce defects into the repository, contrary to what was found in the literature [21]. Again, this may be due to the branching workflow of this specific organization.

Continuing with the results from the questionnaire, 57% of developers agreed that branch scatter increases the risk of merge conflicts and 30% of developers thought that scatter increases the duration of test and build. Both the statistical results and the questionnaire results are not particularly convincing for branch scatter as an integration debt factor. More than half of the developers did, however, think that it affects merge conflict risk so there may still be some merit to investigate this factor further in future research, perhaps in organizations with different branching workflows and branching strategies.

6.1.2.5 Code churn

Code churn refers to code changed or reworked (additions and deletions) on a file over time. While others may define the term differently, code churn in this study is the absolute value of code delta, i.e. the sum of LOC added and removed. The idea of code churn being potentially involved in integration debt was inspired from two studies [18] [19], where code churn is mentioned as a factor in causing defects. Thus an assumption was set that an increase in the amount of code churn would increase the risks of merge conflicts.

Interview results gave more insights about the developers' opinions of impact on code churn on their CI development. 7 out of 10 developers agreed with code churn being a factor in integration debt while others did not think the same or were inconclusive. A majority of participants believed that when files were changed more

frequently by developers, the risk of merge conflict could increase. While another participant made an interesting comment that "If the code that is being re-written does not affect any other parts of the code, then it is fine. It is only a problem when that code affects other code". This statement made logical sense as frequently modifying a piece of code that is highly dependent or affects other parts of the code, are more likely to cause merge conflicts. It may also suggest that factors (such as LOC, NOF and code churn) that are related to making changes to the code base, could potentially have some relationship with dependency, or that dependency as an integration debt factor should take other secondary factors into account. It was not done in this study as dependency was not measured quantitatively which is explained in the section for dependency.

The data mining and analysis showed a moderately strong correlation (rho = 0.559) between code churn and merge conflicts. An interesting observation to make is that the correlation value is very similar to the correlation value that LOC added and LOC removed produced when compared against merge conflicts. Presumably, this is because code churn is essentially LOC added and LOC removed added together. This suggests that perhaps code churn should be considered alongside LOC added and LOC removed as a part of the lines of code factor rather than its own separate factor.

In the questionnaire, participants were given the statement, *higher code churn increases the risk of merge conflicts*. 83% agreed with the statement, while 13% were indecisive and only 3% did not agree. In general, the findings from investigating code churn indicate that a developer can increase the risk of merge conflicts by modifying a file (in terms of LOC). Both the views of the developers and the data from the CI-system point to this realization.

6.1.2.6 Code dependency

Code dependency is a measure of how connected modules are in a software system, although in this study the definition of dependency is expanded to any piece of code that is dependent on any other piece of code. High coupling is often followed by low cohesion within a software system, which means modules in the system that belong together are not grouped together. The idea of code dependency being involved in integration debt originated from reviewing two studies in existing literature [22] [17]. These articles discuss how dependent components need to be synchronized, otherwise, the risk of integration defects is imminent. It is directly mentioned that code dependency is a causing factor of integration debt, although the article just scratches the surface and does not dig deep into integration debt and what it means.

Apart from literature, code dependency was also brought up during the open-ended interviews with CI-managers. It was mentioned that higher code dependency in a commit could have an impact on causing merge conflicts as well as trigger more test cases in the CI-system, which could potentially increase the duration of the test

& build. The fact that dependency directly causes a response (triggers more tests) from the CI-system makes it especially interesting as an integration debt factor since it not only affects the code and potential issues with integrating the code, such as merge conflicts but it also directly affects the CI-system itself during the test and build. This may however not be the case for all CI-systems which means that the importance of dependency could vary greatly between organizations in terms of its impact on the duration of test and build.

From the interviews, when asked about the impact of code dependency on integration debt, 9 out of 10 participants believed that high coupling would increase integration problems and add up to extra effort and resources required to resolve merge conflicts. It was also brought up that development teams would occasionally have to wait for each others' code to be merged to the master branch, perhaps because of cross-functional dependency between the pieces of code that were being developed. Meanwhile, one participant suggested that dependency between commits could be reduced through increased modularization of the code.

Since the CI-system was found to respond to dependency by triggering more tests, an effort was made to investigate how exactly the CI-system determined the level of dependency of a commit. This information was however not found which made it significantly more difficult to investigate this factor statistically. Even if some existing measure of dependency would have been applied, it would likely not have been the same measure used by the CI-system itself and would thus have not have been fully applicable within the studied CI-system. Therefore, due to the unavailability of relevant data in the CI-system and code repositories, dependency was not statistically measured. It would be of interest for future studies to statistically investigate dependency, especially if it could be studied within a CI-system which uses known criteria of dependency for triggering tests.

However, dependency was still brought up in the questionnaire, since it had been recognized as a potential factor through previous methods of evaluation (i.e. literature review, CI-manager interviews and developer interviews). Participants were given two statements regarding dependency. Firstly, a statement that higher code dependency increases the likelihood of merge conflict. Results showed that 80% agreed with the statement while the rest were mainly indecisive. In the other statement, participants were asked whether higher code dependency in a commit increases the duration of test & build. Findings showed that 70% agreed while 23% were indecisive and only 7% of participants disagreed. In summary, although dependency could not be statistically assessed, the results from all other methods point to it being a significant factor in integration debt. Higher dependency seems likely to have an impact on both integration problems in the form of merge conflicts and also the duration of automated test & build.

6.1.2.7 Code development overlap

Development overlap refers to when two or more developers simultaneously work on the same piece of code. Overlap in development was initially identified as a potential factor in integrating debt, through reviewing literature [20]. In this study, it was found that working in parallel can lead to more problems and defects, especially if the changes being made are within small temporal proximity and the content of the modification interfere with each other. Thus development in parallel was further investigated to assess its involvement in integration debt.

From the interview sessions, 9 out of 10 participants believed that working on the same piece of code from different origins is an integration debt factor. From the semi-structured conversations with developers, several of them acknowledged that large scale development will have overlaps in code and believed that solving merge conflicts is part of their daily job. Another participant pointed to poor communication/awareness of ongoing work in the same domain and believed that if the two or more overlapping origins work more closely together, there would be fewer occurrences of overlap and integration problems.

One point to consider when it comes to development overlap is that, as mentioned by the developers in the interviews, the problems from overlap arise when development is separated across different teams or when there is a lack of communication or cooperation within a team. Depending on the organizational workflow, it may be difficult to meaningfully measure the overlap from within the CI-system since overlap is not necessarily a problem if it is known, communicated and coordinated. For instance, developers may pair-program and then push commits, which have overlapping code, from two different origins. This overlap would likely not have a significant impact on integration problems since the developers worked together. However, this would not be visible from the CI-system data.

Similarly to dependency, statistical evaluation was not carried on development overlap. One reason was because of uncertainty in choosing an appropriate span for the temporal proximity of overlapping commits. This simply means that there was no solid number of what is a time range for two or more versions of a piece of code before it can be considered as overlap in development. The second reason was simply because of the origin of commits, as many of the developers at the company did not use Git branches. The third and last reason for not statistically analyzing development overlap was due to the fact that in many cases it was impossible to identify development overlap, as it was being done in isolation on a local computer.

When participants in the questionnaire were asked whether a higher amount of development overlap increases the risk of merge conflicts or not, 90% agreed with the statement. While 7% were indecisive and only 3% disagreed. The questionnaire results showed a rather strong viewpoint from developers toward development overlap being a causing factor of merge conflicts. Taking all the results into account, despite the lack of statistical analysis, the findings from all other methods suggest

that development overlap is a definite factor in integration debt.

6.1.2.8 Branch activity

Branch activity refers to how much time there is between commits from a branch to its master branch, i.e if the branch is inactive for a long time which causes it to fall behind the master branch. Staying behind the master branch can be caused by delays in integration or a lower frequency of commits from developers. The idea of branch activity being involved in integration debt was originated from the company's claim, that staying behind the master branch appeared to be costly for them. The company stated that branches that have accumulated unintegrated code, disallow other branches (developers) to have access to the latest code. Additionally, it was discovered from literature [21], which investigates the effects of branching on software quality, that branch activity is a factor in causing defects. Thus, branch activity was analyzed to assess its impact on integration debt.

From the open-ended interviews with CI-managers at the company, it became clear that there are three ways of committing new code to the repository. Developers can merge code from their personal Git branch, their Git team branch or their local computer where a developer works on a local computer and does not use Git branches. The CI process is displayed in figure 5.1, which gave an overview of how development is done across branches in the project. By realizing that not all developers use branches, it became rather difficult to statistically measure the correlation value of branch activity. This was accompanied by an overlap in the definition between branch activity and frequency of commits as was discussed in the section for frequency of commits. However, there was still a simple calculation done on the data to get the average frequency of commits which is explained later on.

From the interview session, when participants were asked about how often they pull/push their code to the master branch on average, different answers were received. How often they committed vastly varied among participants, as some committed on a daily basis while others would commit once every third week.

As was discussed in the frequency of commits section, there is a consideration to be made about committing more often and thereby causing more commits to be merged which could increase the occurrences of merge conflicts (and spend CI-system resources), or committing less often and thereby falling behind master which increases the risk of merge conflict once the code is committed. Branch activity, frequency of commits and the rate of development (how much code is written) seem to all be related to each other, and perhaps could be melded together into one factor or metric. This could be a research question for future studies.

The results from the questionnaire showed a rather strong conviction from the participants in that low branch activity increases the risk of falling behind the master, which in return could increase the risk of merge conflicts. The result from the

questionnaire showed that 77% of participants believed branch activity to have an impact on integration debt while 23% were indecisive and more interestingly, no one disagreed. To summarize the investigation into branch activity, despite an inconclusive result from the interviews and a lack of statistical analysis of data, the other findings seem to indicate that to some extent, branch activity does have an impact on integration debt level. The accumulated debt in terms of unintegrated code could build up if developers do not commit regularly in accordance with CI principles. The factors of branch activity and frequency of commits could also perhaps be redefined into a more complex metric as discussed before in the sections for these factors.

6.1.2.9 Branch depth

Branch depth can be referred to as the number of sub-branches in a project or distance from sub-branch to the master branch. Similarly to branch activity and branch scatter, literature [21] gave the idea of branch depth being involved in integration debt. In this article, the effect of branching on software quality is measured and it is suggested that an increase in branch depth in a project could increase the number of integration defects.

To further evaluate the effect of branch depth on integration debt, interview sessions were done. The findings were pretty inconclusive, mainly because only some developers tend to use branches, but also the ones who did use branches avoided sub-branches as they believed a high number of sub-branches makes development messy. Interview results showed that 3 out of 10 participants argued that branch depth is a factor, 1 out of 10 disagreed while the rest of the participants were not sure of involvement of branch depth as they did not have any personal experience with branch depth. Branch depth, like the other branching factors, was something that the developers did not have strong opinions on.

As a potential integration debt factor, branch depth may affect integration problems due to it causing developers to be further away from the master branch. This is similar to branch activity with the difference being that activity is measured in time (since the last commit) to master branch whereas branch depth is the distance (in terms of branches) to master branch. Both these factors may cause developers to fall behind the master, and being behind the master may increase the risk of merge conflicts. An interesting point to consider is that higher branch depth requires the code to be merged in several stages, to the branches above, until reaching the master branch. This may increase the time it takes to integrate code and it introduces more possible points where a merge conflict could happen. On the other hand, it provides an opportunity for more frequent merges and potential problems could be discovered earlier which in turn might reduce merge conflicts on the master branch. This dynamic, within the domain of integration debt, could be an interesting topic to investigate in future research.

Branch depth could not be statistically measured reliably. This was due to devel-

opers workflow in the company, as the majority of them did not use sub-branches in development. In the questionnaire, participants were given the statement that high branch depth increases the risk of falling behind the master branch. 30% of developers agreed with the statement, while 23% were indecisive and 7% disagreed. Even questionnaire results were more or less affirmative that branch depth cannot be confidently confirmed as a factor in integration debt. This is primarily because developers did not seem to use sub-branches at all and there is a lack of support to consider branch depth as a causing factor of falling behind the master branch.

6.1.2.10 Duration of code review

As mentioned earlier in this report, Gerrit code review acts as a gate before a commit is being submitted to the CI-system. The purpose of Gerrit is for developers in each development team to review each others' code before an attempt to merge it to master branch. During the open-ended interviews with CI-managers, the idea was brought up of how the code review process could affect integration debt. One assumption was that the more lines of code in a commit, the more time it could potentially take for a developer's teammates to review the code and this would ultimately add up to falling behind the master.

Due to a lack of existing data in the CI-system regarding the duration of code reviews, it was impossible to statistically measure the impact of the code reviews on integration debt. Although developers' opinions on the matter were gauged through interviews and the questionnaire. From the interview 4 out of 10 participants believed that the time taken to review code in Gerrit is a factor in integration debt. Meanwhile, 4 out of 10 did not believe the same and the rest were not sure. Most participants thought of Gerrit as a valuable stage and part of their regular development routine. Even while reviewing a commit with high LOC could take a long time to review, one participant argued that without Gerrit review, more commits are likely to be rejected from the CI-system and that also means more consumption of CI resources.

Continuing on with this train of thought, it might be that code review duration is not necessarily a negative factor in integration debt but a positive one in that if more time is taken for reviews, it may reduce problems with the code which in turn reduces the duration of integration, and possibly also the risk of integration problems. Of course, there would be a limit for when the length of a code review stops being productive and becomes a hindrance.

Another point to consider is that there would likely be significant outliers when trying to measure code review duration as an integration debt factor. Reasons why include the possibility that certain commits are left behind, or that a certain developer leaves the organization and thus the code review will never be resolved. Such cases would produce massive outliers which could significantly skew any data collected. To avoid problems such as these, perhaps code review should be viewed

as a softer and more abstract integration debt factor which could be measured using organizational or management metrics rather than hard statistics. This is outside the scope of this study but it could be interesting for future research.

From the questionnaire, when participants were given the statement that higher LOC per commit increases the duration of Gerrit code review, 80% agreed while the rest disagreed or were unsure. From the questionnaire result, a majority of participants believed that higher LOC increases the time taken on Gerrit review, although they do not necessarily mean it is a factor in integration debt. Also from the interview, the findings are not convincing and therefore the duration of code review is disregarded, within this study, as a factor of integration debt.

6.1.2.11 Fault handling process

The fault handling process refers to how long a developer puts into debugging and fixing issues after a code review or after feedback from the CI-system indicating that defects have been detected within a commit. It was assumed that the time developers allocate to carrying out fault handling could be a factor in falling behind the master branch, and thus integration debt. Although from the interviews, strong findings rejected this assumption. 8 out of 10 participants did not believe that the fault handling process is a factor with 2 out of 10 thought otherwise. Most developers argued that debugging and fixing issues within code is one of their main daily tasks and therefore they did not think it should be a factor in integration debt. For this reason, fault handing was disregarded and not evaluated further.

6.1.2.12 CI-system's factors

In this section factors that are related to the CI-system is discussed. These factors were not discovered from literature but they were all brought up during open-ended interviews with CI managers and developers in interview and questionnaire. CI-system factors are minor factors within the CI-system that could fall under the primary factor of test/build duration. These factors may have an impact on the duration of automated test & build in a project and prolong the time taken for a commit to get merged to the master branch. They were not considered significant enough to be standalone factors but go under test/build duration. Furthermore, these factors were not statistically measured, although there were other findings on these factors which will be interpreted in this section.

- **Fault slip through**

  Fault slip through in simple words can be described as a defect within the code of a commit that should have been discovered in earlier test stages but was only caught later, if at all. When defects pass through a stage where

they should have been detected, they could be identified by other stages of automated testing but in the worst case, the fault remains undiscovered and the code gets merged to the master branch. This is when the real impact of fault slip through can be seen. Therefore it is more cost-effective to detect faults in earlier phases.

Fault slip through was first mentioned during interviews with CI managers, where it was brought up as a potentially interesting factor in integration debt. Beyond that, during the discussions with developers during interviews, it became more clear that fault slip through may affect the duration of automated test & build and merge conflict. In the questionnaire, when developers were asked whether fault slip throughs increase the risk of integration problems, the findings were remarkable. 93% agreed while 7% were indecisive and no one disagreed. The majority of participants believed that fault slip throughs have an impact on both merge conflicts and prolonged duration of test & build. Despite the lack of statistical analysis, the existing findings point to fault slip through being an integration debt factor.

- **Production stops**

  Production stops refer to when there is some problem which causes the CI-machinery to be unable to function properly, and thus production is put on hold since new code cannot be merged. This was mentioned by CI-managers as a potential integration debt factor. In general, there were limited findings about production stops but it could potentially have an effect on the duration of test & build since it would mean that testing and building has to wait until production resumes, and consequently it may increase integration debt levels.

- **Commits getting re-grouped or re-tested**

  During the open-ended interviews with CI-managers, one potential integration debt factor that was discussed was that commits in a CI-system may go through the system, or parts of the system, multiple times. For instance, if an error is found at one stage and the commit is sent back to an earlier stage. Depending on the system, it may also mean that they need to go through the tests again or that they get re-grouped with other commits, starting the procedure over again. This would mean that the duration of test and build increases. Despite a lack of data, reasoning about this factor would indicate that it could affect integration debt because of the increased duration of test and build. Although the magnitude of the impact and to what extent it affects integration debt remains unclear.

- **Availability of CI resources**

  The availability of continuous integration resources refers to the infrastructure used in a testing environment and whether it is sufficient to the scale of

development. Similarly to other CI factors, findings here were limited. However, several developers from the interview believed that the efficiency of CI resources could have an impact on integration debt, as it can affect the duration of test and build. When there is a peak of traffic in the CI, often CI resources perform slower. Thus more resources could mean more commits being simultaneously tested and that reduces feedback time and improves time to integration.

6.1.3 Mapping of integration debt factors

When attempting to answer RQ2, that is built upon findings of RQ1, it became clear that there were different levels of factors (i.e. factors and sub-factors) and thus, there needed to be some kind of an overview of the involved factors in order to visualize their impact on integration debt. That is why the initial factor map (see figure 5.4) was created which was only based on the results of interviews and literature review. The finalized mapping figure shown in the summary of results (5.37), is based on the findings from all the adopted data gathering methods.

In the final version of the mapping figure (5.37), several factors were removed from the figure, due to enough evidence that rejected their impact on integration debt. Additionally, some readjustments between the relationship of factors were done to correctly display the relevance between lower-level factors on their respective higher-level factor. (e.g. secondary factors that cause another primary factor/s). In other words, the idea behind the relationships of factors was to represent what factors affect the factors on the level above them.

The origin of different levels of factors was from distinct data gathering methods. The core factors of the mapping figure (i.e. integration delay, integration workload, and infrequent integration) were from the company's claims where they expressed their problems that caused integration debt in their project. Primary factors (i.e. duration of test/build, merge conflict and falling behind master) were all originated from open-ended interviews with CI managers, where they believed are the cause of the core factors on the level above them. The secondary level factors are the factors that can be considered as measures of integration debt and their overall impact together embodies the integration debt level. In the mapping figure, the primary factors are the causes of primary factors (e.g. LOC per commits on causing merge conflict).

Worth noting is that the primary factors were still measured but the secondary factors are the ones that were viewed as contributing towards the integration debt level. This is because, as mentioned in the results section for the initial figure mapping, the primary factors are the ones that represent integration problems that directly lead to the core factors of increased time, effort or cost. As such, measuring a primary factor, such as merge conflicts, means that it is measured once the merge

conflicts have already happened. It is thus more interesting to measure its sub-factors to be able to assign some risk to merge conflicts happening due to the level of the secondary integration debt factors.

During the analysis of factors, several factors were labeled as inconclusive (mainly branching factors) and are showed in yellow color in the mapping figure. Since not all the developers at the company tend to use branches, it became difficult to statistically measure the correlation of branching factors against merge conflicts or duration of test & build. Although other methods such as questionnaire indicated their positive impact on integration debt. As a result, these factors, i.e. branch scatter, branch depth and branch activity require further investigation to evaluate their impact on integration debt. As a result, due to a lack of concrete evidence to reject/approve these factors, they are still considered as a potential integration debt factor.

One of the weaknesses of doing the mapping in this way is that there is some degree of subjectivity involved and our understanding of the domain is not perfect and perhaps exclusive for only the company involved. However, the mapping figure can still be generalized and applied to other large-scale software projects. Another point worth considering is if doing the mapping this way was ultimately necessary, or if there were other alternative approaches to take. To begin with, we recognize that the mapping is a simplified version of reality. For example, lines of code likely has some relationship with all of the other factors (including secondary factors) since code is the basis of what the CI-system handles and what all the factors relate to. Yet in our mapping lines of code only has a relationship with a few primary factors which it generally affects. For example, the number of lines of code affects code review duration, which is not necessarily always true but is assumed to be true in the general case. This simplification of the relationships was done to create a more manageable overview of the factors and their sub-factors. All in all, the mapping figure could certainly have had a different or expanded hierarchy of factors, however, this study views the relationships between integration debt factors as shown in figure 5.37.

6.1.4 Threshold

To answer RQ3, that is to determine the threshold for integration debt factors, LOC in commits was selected and assessed using a unique approach presented in results, in figure 5.6. Due to this study having a limited time span, only one factor (LOC) was evaluated in terms of thresholds and for a holistic threshold value of integration debt, the rest of the affecting factors are left to be measured. This study does, however, provide an example of how the threshold levels can possibly be measured for other factors as well. The factor of LOC was chosen as it is one of the most fundamental and strongest factors of integration debt.

Before discussing the findings of this threshold part, it is important to keep in mind that the suggested threshold level is project-specific and may be different for other projects. The data used to measure the threshold is the company's repository consisting of two different data sets where each is from a different time span (one 6 months of data and other 2 months). The goal behind setting a threshold level was to be able to predict the risk of integration problems in terms of that factor. In this scenario that means determining the risky amount of lines of unintegrated code that could have a higher likelihood of causing merge conflicts. By knowing the threshold level, the risk of merge conflict occurrence could potentially be mitigated by taking appropriate actions to mitigate the accumulated debt. In other words, the approach used to calculate the threshold was to simply look at the ratio of merge conflicts in each category of LOC. The idea was to be able to predict when there is a high risk of merge conflict in terms of the lines of code in commits. This would have allowed, to some extent, to see which values of lines of unintegrated code is likely to cause merge conflicts.

As shown in figure 5.6, when the grouped LOC data were plotting, it became clear that there was a significant jump in the ratio of merge conflicts (of total commits) when LOC went above 1000. Therefore, the threshold level was chosen to be commits that have above 1000 lines of unintegrated code. Additionally, to get to a more accurate threshold level, the commits between 0-3000 LOC were further investigated as the data was mainly populated in that interval. However, the findings appeared to be inconclusive as no solid threshold could have been set. Thus, the final threshold value is set to be above 1000 LOC for both added and removed lines of code.

Furthermore, the evaluation of the threshold involved using the new data set (shown in section 5.7.1. The grouping and plotting the data in the same way as done for the previous data set. The idea was then to see if that same pattern, that the threshold was set at, exists for the new data set as well or not. If it did then it would have indicated some potential usefulness of being able to predict when an integration debt factor becomes a problem in terms of LOC. However, the results of the evaluation of the threshold were at best inconclusive. It was not possible to discern the same pattern and in fact, the merge conflict occurrences for the new data set did not look very similar to the previous data set in general. If any conclusion is to be drawn from this limited inquiry into the possibility of assigning a threshold for an integration debt factor, it is that it may be very difficult to do so and may also require a significantly larger amount of data than what was used in this study.

6.2 Reflections

6.2.1 Methodology used

Design science was the used research methodology in this study which was carried out in a collaboration with a company in the industry. A design science approach was fitting due to the purpose of the study where the focus was on investigating integration debt, and therefore needing to design an artifact for the factors, metrics, models that were used to define, measure and describe integration debt. It was done together with a company to allow for data to be gathered from a repository of a real-world project as well as being able to get insights from the company's developers and CI-managers.

An initial consideration was to use a case study research methodology instead of design science since the study was in collaboration with a company that would have provided the opportunity to do so. It would have meant that integration debt was investigated using similar methods that were used (e.g. surveys) but without designing and evaluating an artifact. A case study could perhaps have been less solution-oriented and more exploratory. That is ultimately why design science methodology was selected which allowed integration debt that was a new term, to be defined and measured.

The data gathering methods and the data analysis methods used were a mix of qualitative, with literature review, surveys and interviews, and quantitative with data mining and statistical analysis. The qualitative methods were necessary since the integration debt term is not well-established, as such it needed to be explored in a broader sense before it could be investigated in depth. The quantitative analysis could not have been conducted without having an idea of what to quantitatively analyze. This is also the reason why the study was structured in such a way where the quantitative methods only start to be used in iteration 2, with iteration 1 involving qualitative methods and literature review.

6.2.2 Applicability of findings

This study provided a foundation for integration debt. It provided a definition of the term, a set of factors involved in integration debt and a mapping of their relationships. Its main result and its main usefulness was the establishment of an initial understanding of the domain of integration debt. It is essentially a basis for further investigation into integration debt both from an academical perspective as well as being able to meaningfully monitor and predict integration debt levels in industrial projects.

An understanding of integration debt and what it is caused by could lead to several potentially useful applications. Firstly, it allows for the concept of integration debt to be applied to projects and organizations, along with their CI-systems and repositories. It enables organizations, such as software companies in the industry, to understand what level of integration debt they may have and why. It provides some measure of the risk of integration problems. If applied to different repositories or branches, it can also show which parts of a project has more or less integration debt which may prove helpful when attempting to prevent integration problems.

Integration debt could also be measured from bottom-up instead of top-down, meaning that it could be measured from the point of view of individual developers rather than from the master down to branch level. The usefulness with this would be that a developer could see what their level of integration debt is, or how much integration debt they are adding to the project and why. They could then get a sense of their risk of integration problems as well as how to mitigate potential future integration problems.

6.2.3 Difficulties encountered

During the study, some difficulties were encountered which impacted the study in various ways. One such difficulty was the workflow at the company. This proved to be a problem in a few ways. To begin with, the data from the CI-system and repositories were considered sensitive which meant that it could only be accessed and analyzed within the development environment provided by the company and only the tools available in that environment could be used. Another issue with the workflow was in regards to branching strategies, in that many developers at the company did not use branches much at all. This made it very difficult to investigate factors related to branching, such as branch depth.

The CI-system also did not store some kinds of data which would have been useful in investigating certain factors. For instance, it did not store the duration of code reviews which made it difficult to investigate that factor within the time-frame of the study. As a result of the lack of certain kinds of data in the CI-system, some potential integration debt factors could not be investigated in the data mining part of the study (e.g. code dependency).

Coming up with accurate thresholds (for RQ3) and evaluating them in a meaningful way also turned out to be surprisingly difficult. It was often not clear what interval of LOC to use and at which point a threshold could be set. Perhaps it would have been useful to have an even greater quantity of data to be able to see patterns emerge. The data was highly specific to the company and to the development at the company at the time which further increased the difficulty of setting generalized thresholds and evaluating them.

6.2.4 Threats to validity

The threats to validity brought up in this section aim to discuss the trustworthiness of the result. For this purpose, the identified threats were categorized using three main aspects of threats to validity, i.e. construct, internal and external validity (based on guidelines [37]).

6.2.4.1 Internal validity

In regards to the internal validity of findings, there are a few threats to which were recognized and considered when interpreting the results. The first internal threat is regarding the interviews and questionnaire results, where there appears to be some aspect of bias, subjectivity, and ambiguity. This is the case for both open-ended questions and questions such as the ones using the Likert scale in the questionnaire. An interviewee may for example drastically overstate the importance of an integration debt factor because they had problems with it in their personal work. Answers to the Likert scale questions in the questionnaire also had a degree of arbitrariness and subjectivity since there is no objective way of discerning where the line is drawn between, for instance, agree and strongly agree.

Another threat to the internal validity of the study is that we had to put some level of trust in the CI system at the company to provide us with correct data, and it was not possible to verify that all of this data is actually correct. There was no other choice but to assume that the data is correct, and since the company is relying on this data it is not a particularly wild assumption. With that said, during the data mining part of the study, there was some confusion (both from the researchers and CI-managers) about which data is which in the CI-system. It was eventually cleared up by the CI-managers to the best of their ability.

The scope of the investigated factors in integration debt was limited to only the ones that are within the CI-system (which includes repositories and version control). This limitation of scope allowed the potential factors to be measurable and investigable in the study. Other potential factors that were outside the scope of this research were ranged from technical to organizational to psychological aspects of development. However, their involvement was definitely considered on the results but left out. For instance; how different agile teams have their own development routines (e.g. frequency of commits or way of branching) that could affect integration debt. Ultimately, not considering all aspects of integration debt prevented the scope of the study from becoming unreasonably large.

6.2.4.2 External validity

Regarding the external validity of results, the generalizability of the findings is the focus. The study was conducted at a company which had a specific way of code integration, designed to best fit its large-scale software development. Thus when answering RQ1 and RQ2, not all findings could be generalized (e.g. mapping of factors) and therefore when applying the results here in other organizations, further investigation is required. However, the majority of the discovered integration debt factors can be generalized as often there was strong evidence of their involvement in causing integration debt. This is because the factors (e.g. LOC) and their relationships (i.e. higher LOC increases merge conflict risk) are likely part of many software development projects.

As for RQ3, a threshold is highly dependent on the development routines of an organization, because the specific conditions of the organization will likely have an effect on when (at which value) a factor becomes a problem. A large organization with a mature process and well-developed CI-system will, for example, be able to handle a large number of commits before it leads to integration problems whereas a smaller and less mature organization would start struggling at a lower threshold.

The external validity of the study is also threatened by the lack of a more comprehensive final evaluation of the set of factors, the mapping, and thresholds. It would have been useful (although it would be outside of the time-frame of this study) to further evaluate the factors/mapping/thresholds while in use over a longer period of time in order to make sure that they are sensible and applicable in a real-world context.

6.2.4.3 Construct validity

Threats to the construct validity of the results are mainly regarding the methods/measures used in this study. The adopted design science methodology was structured in a way to allow for the collection of data using different methods that could help in answering the RQs. Thus, it is considered that another research study can adopt a different approach investigating integrating debt and its factors. Additionally, the raw data could have been categorized differently, perhaps instead of using weekly data, a longer interval could have been set (e.g. monthly interval) which changes the visualization of the data and to some extent affects the results. This was also noticeable when grouping the data in the threshold evaluation when 100 and 1000 LOC intervals were used. The intervals adopted by this study were arbitrary and it was later noticed that using a different interval (e.g. 500 LOC) could show a different visualization of the findings which may affect which threshold level of integration debt is chosen. Also, determining the threshold level appeared to be specific to the project/organization since the data is specific to that project.

7

Conclusion

This study carried out an investigation into integration debt, its causing factors and how it could potentially be measured. It provides a foundation for the term integration debt which had not yet been investigated. It also gauged the affecting factors of integration debt and looked into how it could be mapped or modeled such that it could aid in predicting the debt level based on its affecting factors. Throughout the research, as more findings were collected, the researchers' understanding of integration debt became broader. This was enabled via an iterative process of data gathering that provided the ability to answer the research questions. In a practical sense, integration debt means how far away (in terms of time, effort, cost) a developer (or a development branch) is from integrating the unintegrated code. It was discovered that the factors involved in integration debt are the potential causes of integration problems within CI-development. This was reportedly costly for the involved company as additional time and resources were required to resolve merge conflicts.

After a detailed analysis of all potential factors, the final approved ones that appeared to have an impact on integration debt were: lines of code, number of files, frequency of commits, code dependency, development overlap and code churn. These were all shown to be definite factors in integration debt. These factors appeared to have an effect on causing merge conflicts and/or affecting the duration of automated test & build, making the code integration process lengthier. Lines of code in commits, number of files and frequency of commits seem to be the most generalizable integration debt factors that are highly influential in integration debt within large-scale software development.

The rest of the investigated factors were either inconclusive or ruled out. The approved factors of integration debt were mapped to understand their relationship to integration debt, and each other. A threshold for LOC was defined and evaluated (on a second data set) with the result that the threshold did not hold for the second data set. The conclusion is that determining thresholds for integration debt factors requires a significant investigation into potential thresholds, and more investigation into integration debt factor thresholds is required in future research.

Furthermore, findings from the interviews and the questionnaire showed that the

continuous integration process and its infrastructure can affect integration debt by prolonging the code integration cycle. The identified CI factors were: availability of CI-resources, fault slip through, production stops and commits getting frequently re-grouped. It appeared that by understanding the notion of integration debt and what can possibly affect it, measures can be taken in order to mitigate the risk of accumulating more integration debt which would have led to more integration problems.

7.1 Future research

To continue the research conducted in this study, there are several potential avenues to explore. To begin with, there is the possibility of further investigating what factors could be involved in integration debt. This study has defined a set of integration debt factors and investigated them through different methods. Future studies could build on this work, both by finding new factors and further validating the previously found factors. Further investigating the affecting factors would not only allow for finding the most generalizable ones that are applicable to software projects but would perhaps also aid in creating a more mature metric that can quantify integration debt as a whole.

Another possible topic for future research would be to further investigate the different relationships between the integration debt factors. The mapping and the analysis conducted in this study could be the first step towards a more complex and more complete model of integration debt and the factors involved, along with their relationships. There is also the possibility to further investigate the factor thresholds. Using this model, the factors, relationships, and thresholds could perhaps be quantified through the development of a program that measures and visualizes the different debt levels of factors in a project in real-time. The tool can alert the user when a threshold is crossed, to mitigate the debt level. A program like this may also have industrial applications since it would allow integration debt to be monitored and analyzed in ongoing projects which may help organizations to further optimize their CI flow.

Bibliography

[1] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 50–60, ACM, 2015.

[2] W. Cunningham, "The wycash portfolio management system," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, (New York, NY, USA), pp. 29–30, ACM, 1992.

[3] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, p. 48–59, 01 2014.

[4] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 426–437, ACM, 2016.

[5] J. Holck and N. Jørgensen, "Continuous integration and quality assurance: a case study of two open source projects," *Australasian Journal of Information Systems*, vol. 11, no. 1, 2003.

[6] G. Booch, *Object Oriented Design with Applications*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991.

[7] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, vol. 122, p. 14, 2006.

[8] A. Debbiche, M. Dienér, and R. Berntsson Svensson, "Challenges when adopting continuous integration: A case study," in *Product-Focused Software Process Improvement* (A. Jedlitschka, P. Kuvaja, M. Kuhrmann, T. Männistö, J. Münch, and M. Raatikainen, eds.), (Cham), pp. 17–32, Springer International Publishing, 2014.

[9] A. Miller, "A hundred days of continuous integration," in *Proceedings of the Agile 2008*, AGILE '08, (Washington, DC, USA), pp. 289–293, IEEE Computer Society, 2008.

[10] P. Kruchten, R. Nord, I. Ozkaya, and D. Falessi, "Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, pp. 51–54, 08 2013.

[11] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498 – 1516, 2013.

[12] T. Besker, A. Martini, and J. Bosch, "The pricey bill of technical debt: When and by whom will it be paid?," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 13–23, Sep. 2017.

[13] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *2014 Sixth International Workshop on Managing Technical Debt*, pp. 1–7, Sep. 2014.

[14] E. Allman, "Managing technical debt," *Queue*, vol. 10, pp. 10:10–10:17, Mar. 2012.

[15] A. Martini, T. Besker, and J. Bosch, "Technical debt tracking: Current state of practice a survey and multiple case study in 15 large organizations," 2018.

[16] G. Singh, D. Singh, and V. Singh, "A study of software metrics," vol. 37, pp. 2230–7893, 01 2011.

[17] M. Staron, W. Meding, C. Höglund, P. Eriksson, J. Nilsson, and J. Hansson, "Identifying implicit architectural dependencies using measures of source code change waves," in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 325–332, Sep. 2013.

[18] G. A. Hall and J. C. Munson, "Software evolution: code delta and code churn," *Journal of Systems and Software*, vol. 54, no. 2, pp. 111 – 118, 2000. Special Issue on Software Maintenance.

[19] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, (New York, NY, USA), pp. 284–292, ACM, 2005.

[20] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large-scale software development: An observational case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, pp. 308–337, July 2001.

[21] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 301–310, Sep. 2012.

[22] R. O. Rogers, "Scaling continuous integration," in *Extreme Programming and Agile Processes in Software Engineering* (J. Eckstein and H. Baumeister, eds.), (Berlin, Heidelberg), pp. 68–76, Springer Berlin Heidelberg, 2004.

[23] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, 2014.

[24] "IEEE Xplore." `https://ieeexplore.ieee.org/`.

[25] "ACM Digital Library." `https://dl.acm.org/`.

[26] "SpringerLink." `https://link.springer.com/`.

[27] "Google Scholar." `https://scholar.google.com/`.

[28] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 275–284, Sep. 2011.

[29] J. Saldana, *The coding manual for qualitative researchers*. SAGE, 2 ed., 2013.

[30] "Gerrit Code Review." `https://www.gerritcodereview.com`.

[31] "GIT, distributed version control system." `https://git-scm.com`.

[32] "RStudio." `https://www.rstudio.com/`.

[33] S. D. Bolboac and L. Jäntschi, "Pearson versus spearman, kendall's tau correlation analysis on structure-activity relationships of biologic active compounds," 2005.

[34] I. Weir, "Pearson's correlation." [Online]. Available: http://www.statstutor.ac.uk/resources/uploaded/pearsons.pdf.

[35] E. James, "Straightforward statistics for the behavioral sciences," Pacific Grove: Brooks/Cole Pub. Co., 1996.

[36] "Jenkins." `https://jenkins.io`.

[37] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, pp. 131–164,

Apr. 2009.

A

Appendix

A.1 Interview guideline

Interview process

- **Question 1:**
  Can you quickly introduce yourself and tell us which team do you work with?
  How long have you worked in the industry?
  What are your current and past roles?

- **Question 2**
  Are you familiar with continuous integration?
  How much work experience do you have with CI?
  What does CI mean to you? (how would you describe it?)

- **Question 3**
  When coding, How do you work with branches? do you work on personal or team branches?
  [Follow up if they work with personal or team branches] After feature development on a branch is completed and the branch is merged into the master, what happens to the branch?

- **Question 4**
  Are there any principles / guidelines or standard practices at your project that you have to follow in regards to CI?

- **Question 5**
  What are some of the common problems/issues when working with CI in your project? What about CI in general, are there any problems when using CI that are worth mentioning?
  [Follow up if there were any issues] How do you usually solve these problems?

- **Question 6**

How often do you integrate your code to master branch? In order words, how often do you commit to master branch?

- **Question 7**
  How often do you pull from the master branch?

- **Question 8**
  What do you think the term integration debt means?
  [if the participant is clueless, introduce the concept]

- **Question 9**
  What do you think could be the potential factors involved in integration debt?

- **Question 10**
  How do you think integration debt factors can be measured?
  Do you think it would be beneficial to measure ID?

- **Question 11**
  According to literature and our understanding of the issue, we have identified several potential factors that could possibly have an impact on integration debt levels. We will go through each of them to hear your opinion to whether they Could affect integration debt in any way or not. And if so, how and why?

  - LOC added/removed per commit
  - Code churn, *i.e. amount of code reworked (additions/deletions) on the same file over time*
  - Frequency of commits
  - Number of files/modules added or removed per commit
  - Time taken to resolve the merge conflicts
  - Developers working on the same piece of code
  - High level module dependency (coupling)
  - Longer than expected automated test and build process
  - Long duration of code review on Gerrit

  - Fault detection and handling, *i.e. discovering the cause of errors, bugs and fixing them*
  - Branch activity/inactivity
  - Branch scatter, *i.e. how much development branches are spread out*
  - Branch depth, *i.e. number of sub-branches or distance from sub-branch to mainline*

- **Question 12**
  Is there anything you would like to add? Any factors you think we have missed, or something else you want to add to the interview?

A.2 Table of results from interview

Table A.1: Categorization of participants answers regarding impact of the potential factors on integration debt

Factors /Responses	**LOC added /removed per commit**	**Code Churn**	**Frequency of commits**	**Number of files/modules added or removed per commit**	**Time take to resolve a merge conflict**
Yes	9	7	10	4	9
No	0	1	0	0	0
Somewhat	1	1	0	6	1
No comment	0	1	0	0	0

Factors /Responses	**Working on the same piece of code**	**High coupling**	**Long automated test/build process**	**Long duration of Gerrit code review**	**Fault handling process**
Yes	9	9	8	4	2
No	1	1	1	4	8
Somewhat	0	0	1	2	0
No comment	0	0	0	0	0

Factors/ Responses	**Branching activity/ inactivity**	**Branch scatter**	**Branch depth**
Yes	3	4	3
No	2	2	1
Somewhat	1	1	2
No comment	4	3	4

A.3 Repository mining code

Example of code for getting specific repository data:

```
1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.File;
4  import java.io.FileReader;
5  import java.io.FileWriter;
6  import java.io.IOException;
7  import java.io.InputStreamReader;
8  import java.util.ArrayList;
9  import java.util.List;
10
11 //Program for getting LOC added/removed for commits in git given their change IDs
12
13 public class getLOCfromChangeID {
14
15         public static void main(String[] args) throws IOException {
16
17                 //List for change IDs and which ones have been checked
18                 List<String> idList = new ArrayList<String>();
19                 List<String> checkedList = new ArrayList<String>();
20
21                 //Read all the change IDs and store them in the list
22                 File f = new File("/home/User/Documents/changeid.txt");
23                 BufferedReader br = new BufferedReader(new FileReader(f));
24                 try {
25                         String id = "a";
26                         while(id != null) {
27                                 id = br.readLine();
28                                 idList.add(id);
29                         }
30                 }finally {
31                         br.close();
32                 }
33
34                 //Writers and lists to output the LOC added/removed
35                 BufferedWriter bw = new BufferedWriter(
36                 new FileWriter("/home/User/Documents/locAdd2.txt"));
37                 BufferedWriter bw2 = new BufferedWriter(
38                 new FileWriter("/home/User/Documents/locRem2.txt"));
39                 List<String> locAdd = new ArrayList<String>();
40                 List<String> locRem = new ArrayList<String>();
41
42                 //for each change ID
```

```
43                for(int i2 = 0;i2<idList.size()-1;i2++){
44
45                        //git command to get each commit for a specific change ID
46                        String cmd =
47                        "cd \'/home/User/Documents/repo/
48                        && git log --all --stat --grep='Change-Id: ";
49
50                        String idfromfile = idList.get(i2);
51                        idfromfile = idfromfile.split("-")[0];
52
53                        idfromfile = idfromfile + "'";
54                        cmd = cmd + idfromfile;
55
56                        //Avoid checking the same commit again
57                        if(checkedList.contains(idfromfile)){
58                                continue;
59                        }
60                        checkedList.add(idfromfile);
61
62                        //Running the git command
63                        ProcessBuilder builder = new ProcessBuilder(
64                            "sh", "-c", cmd+"");
65                        builder.redirectErrorStream(true);
66                        Process p = builder.start();
67                        BufferedReader r = new BufferedReader(
68                        new InputStreamReader(p.getInputStream()));
69                        String line;
70
71                        //Look for LOC in the lines of the returned commits
72                        while (true) {
73                            line = r.readLine();
74                            if (line == null) { break; }
75                            System.out.println(line);
76                            if(line.contains("insertions")){
77
78                                //Get LOC added and removed, and add to lists
79                                    String[] s = line.split("\\s+");
80                                    locAdd.add(s[4]);
81                                    locRem.add(s[6]);
82                            }
83                      }
84                }
85                //Write the LOC added to file
86                for(int i = 0;i<locAdd.size();i++){
87                        bw.write(locAdd.get(i)+"\n");
88
```

```
89              }
90
91                  //Write the LOC removed to file
92                  for(int i = 0;i<locRem.size();i++){
93                          bw2.write(locRem.get(i)+"\n");
94
95              }
96
97                  bw.close();
98                  bw2.close();
99              }}
```

A.4 Code for organizing repository data

Example of code for organizing data for analysis:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;

//Program for organizing LOC data into segments of 1000 LOC each
//where the number of commits in each segment is counted

public class LOC_counter {

        public static void main(String[] args) throws IOException {

                //Read the files with the data
                File f1 = new File("/home/user/Documents/LOC_merged.txt");
                File f2 = new File("/home/user/Documents/LOC_mergeconflicts.txt");

                //Prepare to write the output
                BufferedWriter bw =
                new BufferedWriter(
                new FileWriter("/home/user/Documents/LOC_segmented.txt"));

                //Call the LOC method which segments and writes the data
                LOC(f1,bw);
                LOC(f2,bw);
                bw.close();

        }
        public static void LOC(File f, BufferedWriter bw) throws IOException{

                //Reader and map for segments
                BufferedReader r = new BufferedReader(new FileReader(f));
                HashMap<String, Integer> seg = new HashMap<String, Integer>();
                String line;

```

```
43                  //Loop for running through each commit in the data
44                  while (true) {
45                      line = r.readLine();
46                      if (line == null) { break; }
47                      int locline = Integer.parseInt(line);
48
49                      //Round to nearest 1000 to find which segment it belongs to
50                      int round = ((locline + 999) / 1000 ) * 1000;
51                      String segkey = ((round - 1000) + " to " + round);
52
53                      //If segment has not been created, create it
54                      if(!seg.containsKey(segkey)){
55                              seg.put(segkey, 0);
56                      }
57                      int segvalue = seg.get(segkey);
58
59                      //Increment the counter for this segment
60                      segvalue++;
61                      seg.put(segkey, segvalue);
62
63                  }
64
65                  //Line for dividing merged vs merge conflict data in the output
66                  bw.write("----------------------"+"\n");
67                  List<String> sortlist = new ArrayList();
68
69                  //Run through the map to add elements to list for sorting
70                  for (String name: seg.keySet()){
71                              String key = name.toString();
72                              String value = seg.get(name).toString();
73                              sortlist.add(key + " " + value);
74                  }
75
76                  //Create comparator object and sort list
77                  segCompare s1 = new segCompare();
78                  Collections.sort(sortlist, s1);
79
80                  //Write the list to output
81                  for(String s : sortlist){
82                          bw.write(s+"\n");
83                  }
84                  }
85  }
86
87  //Comparator implementation for sorting based on LOC segments
88  class segCompare implements Comparator<String>{
```

```
89
90          @Override
91          public int compare(String arg0, String arg1) {
92
93                  int key1 = Integer.parseInt(arg0.split(" ")[0]);
94                  int key2 = Integer.parseInt(arg1.split(" ")[0]);
95
96                  if(key1 > key2){
97                          return 1;
98                  }
99                  if(key2 > key1){
100                         return -1;
101                 }
102                 return 0;
103         }
104 }
```

A.5 Questionnaire questions

Dear participant,

Firstly, thank you for taking part in our questionnaire. Integration debt is based on the concept of technical debt. In continuous integration development, you take on debt when you have unintegrated code which will grow over time and lead to integration problems. Integration debt then is a measure that provides an overview of how far away (in terms of time, effort, cost) you are from integrating your (unintegrated) code. In other words, when software developers stay behind master branch, i.e. not integrating local code to master and falling behind it by not pulling often. The more the debt, the higher the risk of integration problems such as merge conflicts.

There are many different factors that could affect your integration debt since anything that affects the time, effort or cost required to merge your unintegrated code could be seen as an integration debt factor. We have identified some of these factors through repository mining of the project, reviewing existing literature and a round of interview with some of the consultants. At this stage of our study, we would appreciate your input to validate or invalidate these factors. We have designed a set of 19 questions and based on your experience, we would like you to know to what extent you agree with the following statements. The metric used for measuring the scale is the Likert scale with 5 multiple choice options.

1. Higher lines of code per commit increases the likelihood of merge conflicts.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

2. Higher number of modified files per commit increases the likelihood of merge conflicts.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

3. Higher code churn increases the risk of merge conflicts. (Code churn is a measure of how much the code in a file changes; defined as sum of added, modified and deleted lines over time.)

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

4. Higher amount of development overlap (two or more developers working on the same piece of code) increases the risk of merge conflicts.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

5. High branch scatter (how much development is spread out across different development branches) increases the likelihood of merge conflicts.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

6. Higher code dependency (high coupling) increase the likelihood of merge conflicts.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

7. What are other factors that you think might affect the risk of merge conflicts?

[Open answer]

8. Higher code dependency (high coupling) in a commit increases the duration of test and build.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

9. High branch Scatter (how much development is spread out across development branches) increases the duration of test and build.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

10. Higher number of modified files per commit increases the duration of test and build.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree

☐ Agree
☐ Strongly agree

11. Higher lines of code per commit increases duration of Gerrit code review.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

12. Higher lines of code per commit increases the duration of automated test and build.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

13. What are other factors that you think might affect duration of test/build and code review?

[Open answer]

14. High branch depth (Number of sub-branches, or distance from sub-branch to master) increases the risk of falling behind master branch.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

15. Low branch activity (how often a developer pulls and pushes code) increases the risk of falling behind master branch.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

16. Low frequency of commits increases falling behind master which in turn increases the risk of merge conflicts.

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

17. Do fault slip throughs (faults that remain undetected until a later stage) increase the risk of integration problems?

☐ Strongly disagree
☐ Disagree
☐ Neither agree nor disagree
☐ Agree
☐ Strongly agree

18. What factors increase the risk of fault slip throughs?

[Open answer]

19. Is there anything you'd like to add to what does or does not affect integration debt?

[Open answer]