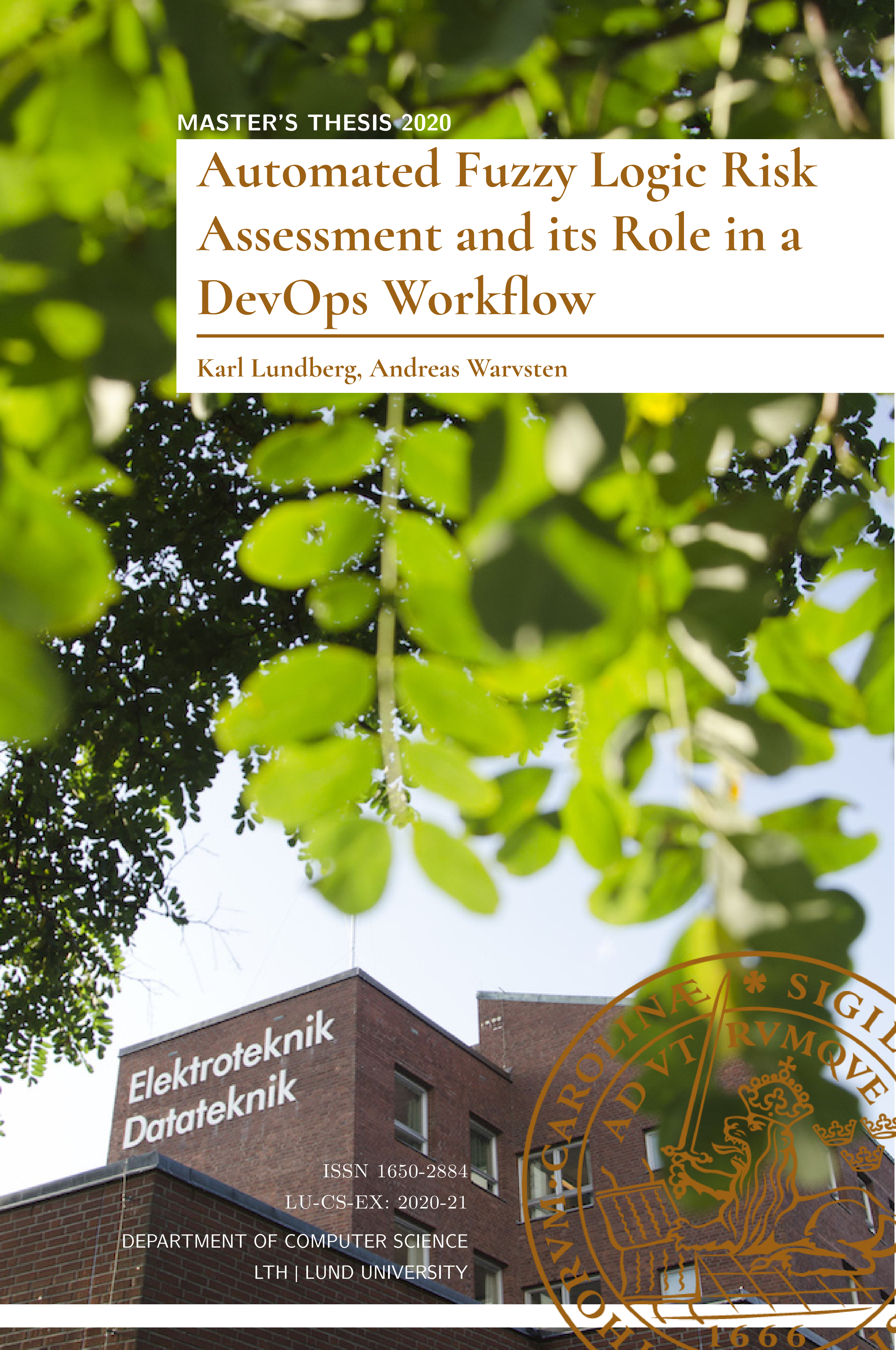


MASTER'S THESIS 2020

Automated Fuzzy Logic Risk Assessment and its Role in a DevOps Workflow

Karl Lundberg, Andreas Warvsten

ISSN 1650-2884
LU-CS-EX: 2020-21

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-21

Automated Fuzzy Logic Risk Assessment and its Role in a DevOps Workflow

Karl Lundberg, Andreas Warvsten

Automated Fuzzy Logic Risk Assessment and its Role in a DevOps Workflow

Karl Lundberg
ine15klu@student.lu.se

Andreas Warvsten
ine15awa@student.lu.se

June 17, 2020

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

Supervisors: Jonas Schultz, jonas.schultz@sinch.com
Ulf Asklund, ulf.asklund@cs.lth.se

Examiner: Martin Höst, martin.host@cs.lth.se

Abstract

Risk management is a key component of the DevOps philosophy. In contrary to the general paradigm of making data driven decisions, it is an area that largely is based on subjective predictions and gut feel of involved competence. One of the benefits of a DevOps workflow that includes versioning tools, is the metadata stored during the development process. This paper studies if this metadata can be used in order to automate some parts of the risk assessment process, and to what extent subjectivity can be removed from the process.

Keywords: Risk management, Risk Assessment, DevOps, Machine Learning, Git

Acknowledgements

First of all, we would like to thank Jonas Schultz, our supervisor and contact at Sinch AB for the continuous support, and also for conceiving the embryo of the idea around which the project was built.

We would also like to thank the employees at Sinch AB that have been kind enough to lend us their time for interviews and support, even given the unfavourable circumstances spring 2020 has provided us with.

Furthermore, we want to thank Ulf Asklund and Martin Höst of the Computer Science institution at LTH for their support, curiosity, and trust in the idea.

Contents

Chapter 1
Introduction

The current handling of change deployment at Sinch AB includes a risk assessment step based on human estimation. This process is time-consuming, especially for cases deemed critical, in which a committee of senior employees has to review the change before deploying it. An automated risk assessment service integrated into the company's existing ecosystem could potentially alleviate the need for human intervention and render some of these costly meetings unnecessary, thus freeing important resources. An automated process could also potentially reach a higher general precision and expedite the risk assessment process in general. This would be a big step in enabling a culture of data-driven decision making, and further reinforcing the DevOps mindset, allowing the company to better meet ever-increasing customer demands.

1.1 Background

The general idea behind the project was proposed by the Malmö-based software company Sinch AB. Through discussion involving supervisors at the Department of Computer Science at LTH, as well as stakeholders at Sinch, the idea was processed until it met the requirements of a thesis project, while still being of interest to stakeholders at Sinch. There has been a substantial amount of research done on the software risk assessment area, however, the results are often abstract in nature and offers no tangible guidance as to how they would be bountifully implemented. Our hope is that through combining a theoretically heavy part with a more hands-on case study, we will be able to present results that both contribute to the body of knowledge, and offers value creation in Sinch's day-to-day operations.

1.2 Project Purpose

The purpose of this thesis is to investigate whether there are factors available for extraction during the software development process that can be analyzed in a way that facilitates fair risk assessment on a deployment-level. This factor extraction and risk assessment process shall require minimal human intervention and involve minimal human judgment. The result of the risk assessment is meant to function as an integrated last-line-of-defence in the CD-pipeline currently employed at Sinch. It shall be investigated whether the assessment is accurate enough to allow deployments with a low risk score to be automatically deployed to production, whereas ones with a higher score need to be stopped and reviewed manually. In the case that such extensive automation is not feasible, it will be studied how the automated assessment can be used to otherwise facilitate the company's risk management work. It is believed that a more nuanced result than the current single value approach could be of assistance in the reviewing process, and in deciding which competence needs to be involved, and perhaps even more relevant, what competence does not need to be involved.

1.2.1 Research Questions

To summarize, the questions that the project intends to answer can be formulated as:

R1 Are there development-related meta factors that can be extracted from common versioning tools, that can help in assessing risk associated with software deployment?

R2 Can this risk assessment method be utilized to implement an automated last-line-of-defence service integrated into the currently used development pipeline, and how would this benefit Sinch?

1.3 Methodology

To answer these questions, a two-pronged approach was applied. Firstly, a literature study was conducted. The main aim of this literature study was to find which risk factors usually are considered when assessing risk in a software development context, as well as to find common methods of assessing software risk. Tangential areas that proved important during the case study will also be touched upon. Existing risk models and previous studies regarding software risk factors were considered. Methods of assessing software risk, used in the different models, are described and the most prominent and frequently occurring software risk factors are presented.

Secondly, a case study was carried out at the company Sinch AB. In the study, git metadata, found in the largest Sinch project, could be extracted. By then utilizing the so called SZZ algorithm, it was possible to label specific commits as faulty or non-faulty, making it possible to analyze the risk contribution of different commit factors, like number of insertions, files changed, etc. The analysis was done both through manual statistical analysis and by the use of machine learning. The results from this risk factor impact analysis was then harnessed in order to design DevOps-compatible automated risk assessment tool, which is believed to realize business value at Sinch. A more detailed, step by step, description of the case study approach is described in Chapter 3.

1.4 Related Work

Gitrisky

Gitrisky [26] is a commit-level risk prediction tool developed by Henry Hinnefeld. It is based on the metadata available in git and it utilizes machine learning to predict the risk of a commit. The way Hinnefeld defines faulty states and subsequently label bugs is a continuation of the SZZ algorithm, an algorithm that we also concluded worked the best for our use-case. Generally, it can be said that gitrisky had a bottom-up perspective, and started out by figuring out what could be implied from a single commit. We took a top-down approach and tried to figure out what kind of service could help in the context of DevOps and risk management at Sinch. Our results ended up having both a similar process and similar results to gitrisky, something that further solidifies the merits of the approach. However, we expanded on the work and conducted research on what factors seem to have the most influence on bug risk and also evaluated different machine learning algorithms by investigating their performance on projects with varying sizes and purposes. Our tool proved better suited for large size projects and is built with DevOps integration in mind. Furthermore, it considers complete deployments and can produce a risk score on sets of commits, whereas gitrisky only looks at the individual commits.

Reproducibility and Credibility in Empirical Software Engineering: A Case Study based on a Systematic Literature Review of the use of the SZZ algorithm

This study [51] investigates the performance of the bug identification algorithm which is based on the assumption that "a given bug was introduced by the lines of code that were modified to fix it", i.e the same assumption used by Hinnefeld. They outline some common mistakes in studies that are based on this method, and also some limitations of the method itself. The implications of this, and ways to increase the bug labeling accuracy is discussed in section 5.2.

Automation Possibilities in Information Security Management

One of the sub-goals of the thesis project was to study ways of automating risk assessment in order to comply with ISO-27001 standards. Although the standards mostly concern information security, it is believed that risk management is a key component in achieving the certification. *Automation Possibilities in Information Security Management* is a paper that explores what parts of this risk management work that can be automated, to what ends, and what tools would be necessary. The tools mentioned in the paper differ from the ones in use at Sinch, but some of the principles and the ISO-related context pertain to our work, especially in relation to research question R2.

ISO 31000:2009—Setting a New Standard for Risk Management

ISO guide 31000:2009 [49] is a widely used framework for risk management and referenced often in research on the area. This document was used as a frame of reference for the risk management work and activities. Later in the project, we scoped heavily, and our results are more in the scope of risk assessment, a component of risk management. The document was invaluable in providing a birds-eye view of common risk management guidelines and to codify the otherwise rather abstract risk-related concepts.

1.5 Contribution Statement

We have worked in close tandem for most parts of the project, both in a research and writing sense. Regarding the theory chapter, Andreas was slightly more involved in the risk model and risk factor sections, while Karl was slightly more involved in the beginning and end parts of the Risk section, and in the DevOps, Machine Learning and Code Quality Metrics sections. For the investigative part of the case study we worked closely together. Going forwards, Karl was slightly more involved in implementing the factor extraction and bug labeling implementation. The applied factor risk impact analysis brings the first clear split, where Andreas did the manual statistical approach, and Karl was responsible for the machine learning. Our work during findings and discussion is indistinguishable.

Chapter 2
Theory

This section aims to afford the theoretical background needed to understand the discussion in the later parts of the paper, and to lend scientific backing to solutions propositioned. Furthermore, it will help in designing the case study in a way that furthers research rather than just replicates it. The research is interdisciplinary, and areas touched upon include risk, risk management, DevOps, and machine learning. It is hoped that the section will provide the base for what factors to study in order to answer research question R1, and the DevOps philosophies to consider when answering research question R2.

2.1 DevOps and the Continuous Paradigm

The world of software development has undergone a series of paradigmatic shifts of varying magnitudes, the shift from a waterfall-based development mode to an agile one being one of the better known. Some argue that the area is in a transitional phase right now [32]. This shift is caused mainly by the emergence of a new field, DevOps, a branch concerned more with the how, than with the what of software development. Traditionally, there has been, for any medium to large scale software developer, a clear distinction between the development side of things and the operations side of things [3]. DevOps aims to blend these areas in a way that minimizes the overhead created by this schism, thus enabling developers to much more frequently deliver code updates, leading to a less rigid, and more adaptable workflow [29].

To contextualize the results of this report it essential to have an understanding of some of the most central DevOps concepts. Generally, how far the company has gotten in this process can be divided into three different stages [58]: continuous integration (CI), continuous delivery (CDE), and continuous deployment (CD), these will be expanded upon later in this chapter.

2.1.1 Workflows and pipelines

One of the most common measurements taken to shift an organization's processes towards ones closer to DevOps best practices, is defining a workflow. A workflow, according to Dumas et al [15]. consists of a coordinated set of activities that have to be executed in order to achieve a predefined goal. The specifics of this workflow is oftentimes referred to as a pipeline. It is comprised of the series of sub-processes that pertain to a software artifact. This pipeline is often circular, or semi-circular in nature, and encompasses the whole development cycle, from creation to maintenance. A common way of illustrating this continuous pipeline is as a horizontal eight, such as in figure 2.1. How extensive and integrated this procedure is, varies from project to project, and companies can decide what parts of the development process to define a clear framework for. The following figure outlines the eight basic phases of an extensive DevOps pipeline as it is defined in one work [41]. It can be important to notice that while the terms used to denote the phases vary across related work, their meaning is rather consistent.

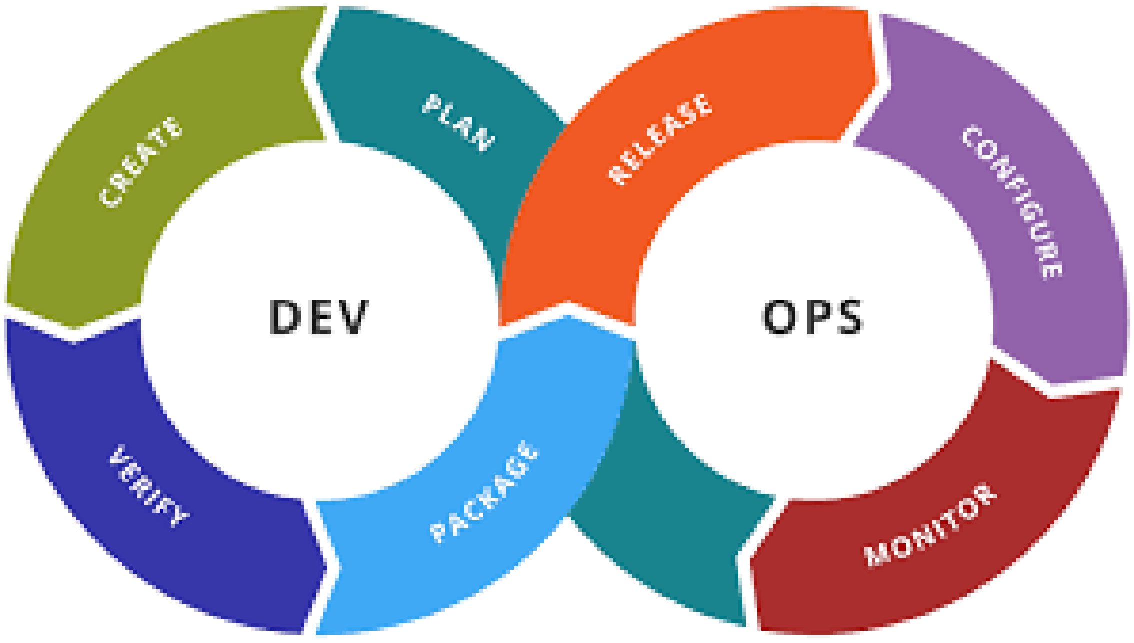


Figure 2.1: The activities in a DevOps workflow and how they tie together [14]

2.1.2 DevOps principles - CAMS

Constructing and implementing a pipeline-based workflow is one thing. Constructing and implementing a pipeline-based workflow that actually benefits the business goals of the organization, is another. This puts demands on the company infrastructure and culture. One way of highlighting the factors that make this possible is the CAMS-model [25], popularized by Damon Edwards and John Willis. This is an acronym that stands for **Culture, Automation, Measuring**, and **Sharing**. The meaning behind these concepts will be expanded upon below.

Culture

The most central part in the DevOps philosophy is that of a strong company culture. This includes a host of best practices, and the company's ability to adhere to these practices [25]. Some common practices include:

- The usage of an issuing framework, such as Scrum, JIRA, kanban, etc. This is a conventional, standardized way of focusing, monitoring, and prioritizing the efforts of a team.
- Monitoring and limiting technical debt. Technical debt is the term for the need of extra work that will occur down the line because of a sub-par, but quick fix [30]. Limiting the technical debt means taking this time into account when choosing a solution, and therefore promotes the implementation of easily maintained ones
- Silo awareness and diffusion. Companies often tend to structure organizations by business function. This model is prone to group over-specialization and that teams tunnel vision on their own tasks, leading to a lack of inter-organizational communication and collaboration. Diffusing, or aligning these silos so that they are working towards a common goal is essential for implementing functional DevOps framework [24].

Generally, the cultural aspect of DevOps needs to be fitted to the specific organizations' needs and capabilities. As such, there exist no one-size-fits-all list of practices that need to be implemented [37]. Every organization needs to be studied, and a framework needs to be tailored specifically for it.

Automation

Historically speaking, building, testing, and deploying have all been time-consuming activities, often performed by a team separate to the one that developed the code [59]. This adds organizational overhead and leads to a longer lead time, something that DevOps as a field works to minimize. Part of an effective solution for this problem is automation [38]. Automation is the process of writing a service or protocol that is able to repeatedly perform a task that used to be handled manually. As such it can have a larger initial time cost, but if done right, reduce the accumulative technical debt. To be able to automate an activity efficiently it is essential that a deep understanding of the underlying processes has been acquired, and that they are all sound and serve a purpose. Automating a flawed process just makes bad processes happen more quickly. Furthermore, it is essential to understand how much of your organization that can be automated. Not every company can realign to a cloud-native code fit for total automation, just because some of the most cutting edge actors are. There are oftentimes going to be legacy code and frameworks where the initial cost of automating the task is going to be higher than the winnings in reduced technical debt. Some particular areas do generally lend themselves very well for automation. Characteristics that are often automated in a DevOps context include [38]:

- Test suites, especially unit tests in a test-driven environment.
- Code deployment. Enabling a short journey from the development environment to a deployment/production one is key for rapid deployment.
- Production metric collection. Feedback is what enables correct decision making. Quick feedback leads to the possibility of quick realignment.

Measure

At the heart of a well-implemented DevOps workflow, is a continuous improvement process [18], or CIP for short. One famous example of such a model is Kaizen (from the Japanese ”Kai”, meaning change, and ”zen”, meaning good). This model states that the key principles are feedback, efficiency and evolution. Key features include that big changes should be a result of several small changes, and come from workers rather than from the top down. It also urges increased ownership over work and focus on individual improvement [19].

To understand whether improvement is taking place it is essential that there are metrics covering the areas of interest. Historical monitoring data might also lead to insights into the development process as a whole and what type of actions have a higher risk of causing issues further down the pipeline [25].

Sharing

The final aspect touched upon by the CAMS model is sharing. There are three key components that are encompassed by the term [30].

- Visibility, the possibility of gleaning insights from separated parts of the organization. The end goal of high visibility is the knowledge of whether there's relevant competence located elsewhere in the organization, and to make sure to avoid double maintenance.
- Transparency, the diffusion of an organization-wide vision and goal. This is key to maintain the alignment of silos so that everyone is working towards a common goal.
- Transfer of knowledge, the goal for any organization should be to increase, and make available as much collective intelligence as possible. Information proliferation is also a key aspect. In order to avoid constraints such as the choke that arises when the one person in the company with the required competence to perform something for example goes on vacation, it is important that critical information is shared.

2.1.3 The different stages of DevOps speed

As mentioned before, there are some distinct stages for how integrated the DevOps mindset is in an organization, and to what extent they have chosen to allow the continuous increments [58]. These stages somewhat loosely defined and can overlap. Generally, however, they can be broken down into three stages, continuous integration (CI), continuous delivery (CDE), and continuous deployment (CD). These concepts will be broken down and explained in this section. A prerequisite for implementing these stages in an organization that already has a strong culture of agile development [30].

Agile

Agile is a mode of development that focuses on maximizing developer productivity through the use of certain practices[6]. It emphasizes, among other things the importance of iterative improvement and cross-functional teams. The most dramatic change from its predecessor the waterfall model, however, is the emphasis of a cyclical development process . An artifact is developed and iterated upon in smaller time-frames often referred to as sprints, whereas waterfall based approaches only go through each stage once. This difference is illustrated in figure 2.2. It is believed that this leads to a quicker value realization for the customer, and a continuous validation process that allows for the end result to be closer to what the customer expected [50].

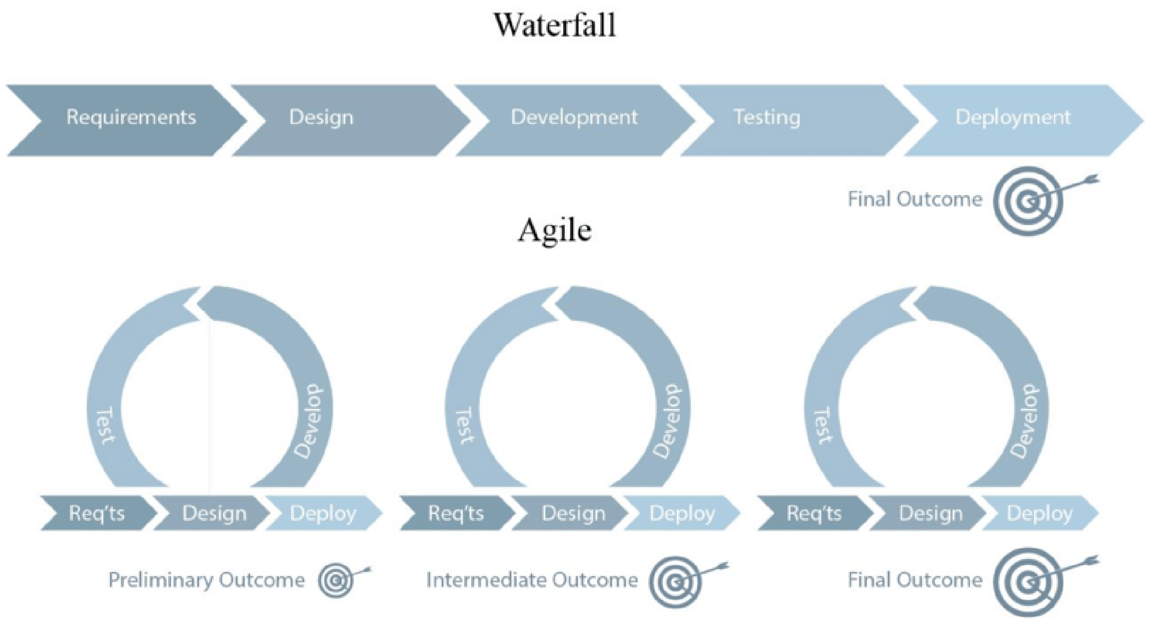


Figure 2.2: The cyclical nature of agile contrasted with the one-dimensional nature of the waterfall model [6]

Continuous Integration

This is the first step in the DevOps ladder. Here the goal is for the developers to push to a common repository as often as possible, preferably several times a day. This method brings several benefits such as increased fault isolation, lower MTTR (mean time to resolution), and lower risk of double maintenance and incompatible solutions [58]. Continuous Integration is most commonly enabled through a development pipeline that includes automated builds and testing. Furthermore, widespread usage of a versioning tool that allows several developers to contribute to a common codebase is also essential [21]. This stage involves the activities enclosed in the blue marking in figure 2.3.

Continuous delivery

When the company has achieved continuous integration, the next step is continuous delivery. This concept aims at not only being able to merge into the main branch as often as possible but also being able to release green, updated code to customers rapidly and repeatedly [58].

To achieve this it is important that not only the build, and testing phases are automated, but also the release process [28]. Releasing green code should be no harder than pressing a button or writing one command in the command line. Benefits of this philosophy include shorter cycle time, smaller batches to search for potential bugs, and quicker feedback on functionality in production environment [58]. The activities involved in this step are denoted as continuous delivery in figure 2.3.

Continuous deployment

This is the most extensive approach. Here the pipeline has to be extensive enough to the point where even the developers can deploy green code instantly. This allows for the best fit to market, shortest cycle time, and quickest feedback. However, it also puts the highest demands on the project infrastructure and on the individual developers [58]. It is important to factor in the risks that accompany this mode of deployment. Developers will make mistakes, and if those are uncaught by building and testing stages of the pipeline they will be deployed into production. Rigorous manual reviews are not conjunctive with the rapid code releases that go with this mode. How critical this increased risk is varies in between projects, and will be studied further in section 2.2, *Risk*.

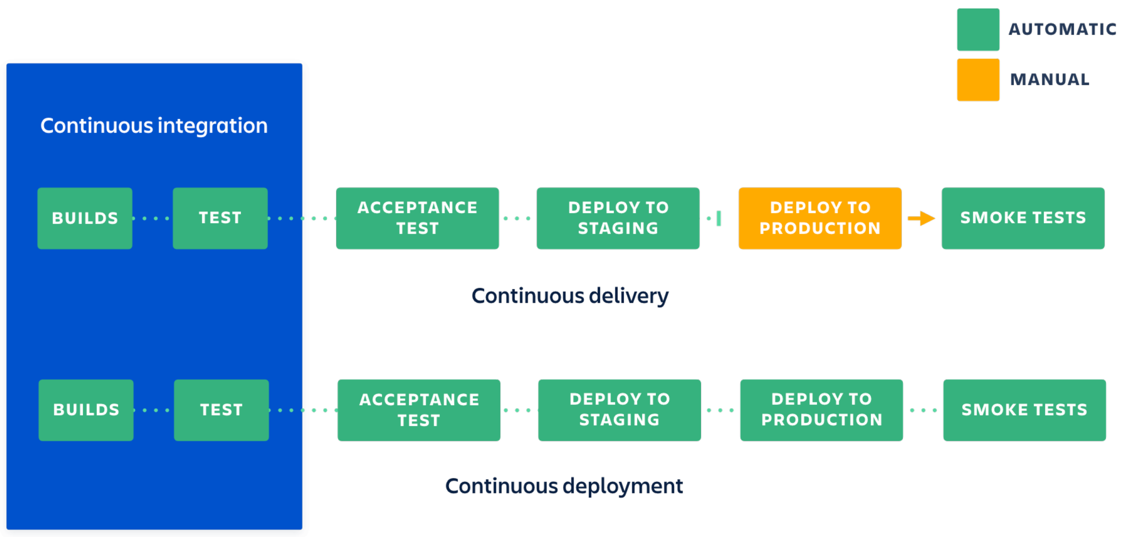


Figure 2.3: The difference between continuous integration, delivery and deployment [58]

2.2 Risk

There has been a vast amount of research done on the subject of risk, in a multitude of different contexts. However, the nature of the phenomena is such that there are no clearly standardized ways of approaching it. This section aims to decipher the body of risk-related work, in order to frame and contextualize our own future research.

2.2.1 Definitions and Terminology

Most people have an intuitive understanding of what risk is. However, in order to study the phenomena from a scientific perspective, it is important that it is defined in a way that allows for analysis. This is not a trivial task. In one of the more commonly referred works on risk [27], Holton states:

> *"The financial literature frequently mentions risk, but it lacks a widely accepted definition. This omission is no coincidence. Risk is an intuitive notion that resists formal definition."*

This section aims to act as a glossary, wherein some abstract concepts are codified and concretized in order to make it clear what is being referred to. Furthermore, it will be explored what factors actually constitute our intuitive notion of risk.

The Two Components of Risk

Holton further explains in his text, that risk denotes an uncertain event that will affect elements, and may occur in some present or future process. For the purpose of this paper, and in the context of software engineering, some additional definitions are made. Risk will be defined as a function of two parts [54], the first one being the *adverse impact* an event would have if it occurred and the second one the *likelihood* of the event occurring. These parts will be further explored in later sections.

Failure, error, fault

In order to further understand risk, it is essential to understand what is considered adverse events. These unfavourable events can be viewed from multiple levels of abstraction and can occur in different stages of maturity. In order to research them, it is important with some codifying. The level of maturity is often split up in three terms: failure, error and bug [43]. These terms are often erroneously believed to be interchangeable, but in the scope of risk management and software testing, they carry significantly different meanings.

- **Failure** is an externally observable malfunction of the program, in relation to specifications or expected behaviour.
- **Error** is a state of the program in which failures are possible.
- **Fault**, is the root cause (in the codebase or environment) of the error state.

In this text, in accordance to conventions in the risk management area, "faults" will be used interchangeably with "bugs", and "errors" with "faulty states".

Bug Classification

One way of categorizing bugs is through separating them as one of four types, based on change magnitude [46]. This is illustrated in figure 2.4. The first and second types are the ones that we intuitively know about, the third and fourth are more complex categories. Type 1 refers to a bug being fixed in one single location (i.e., one file), while Type 2 refers to bugs being fixed in more than one location. In Figure 2, only two locations are shown for the sake of clarity, but many more locations could be involved in the fix of a bug. Type 3 refers to multiple bugs that are fixed in the exact same location. Type 4 is an extension of Type 3, where multiple bugs are resolved by modifying the same set of locations.

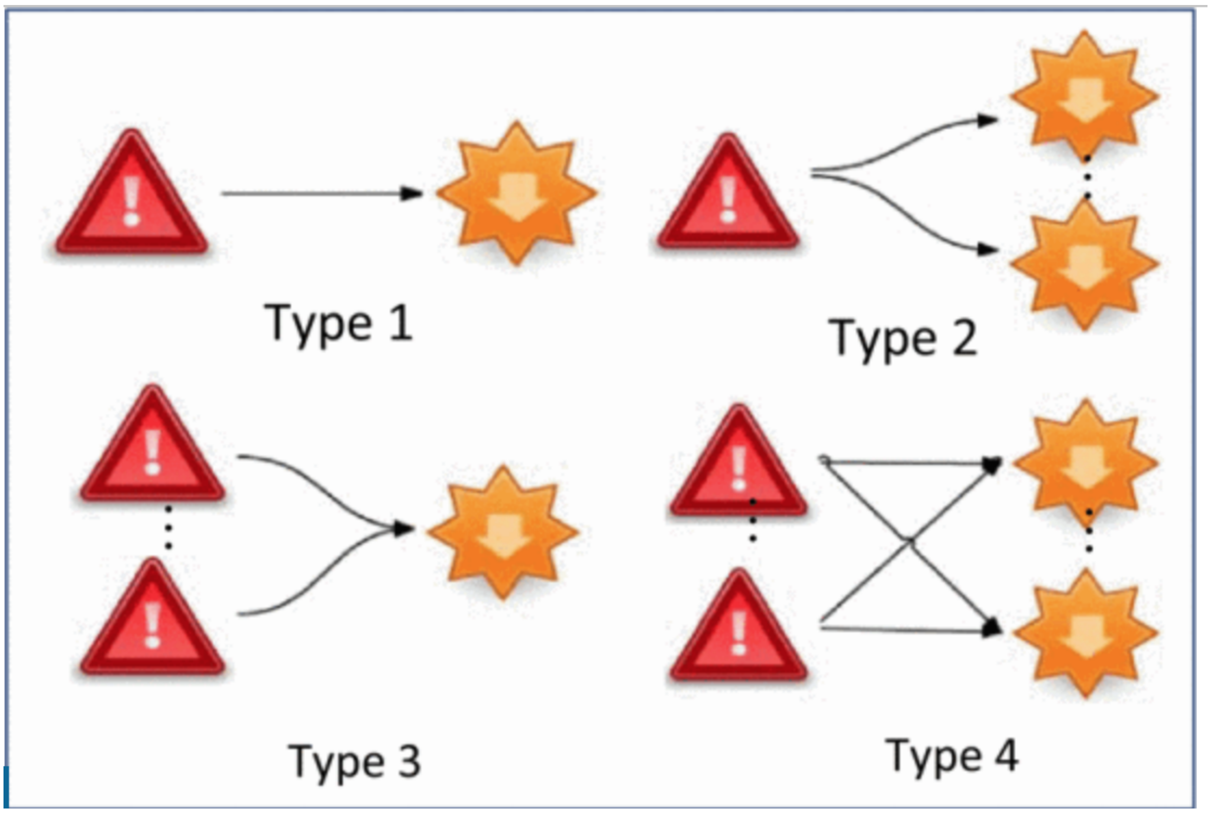


Figure 2.4: The four types of bugs outlined by Nayrolles, Mathieu and Hamou-Lhadj, Abdelwahab [46]

Furthermore, the connection between bug and failure is not an apparent one. In order to study this relationship further, we need a systematic way of establishing this connection. One proposed way is spotting bug-inducing commits in retrospect and tying them to the adverse events. This, however, is far from trivial. The main problem is that neither bug inducing commits or faulty states are naively visible, the program itself does not know that something potentially could go wrong, and if it does, it does not know what caused it. One way around this, that has reasonable scientific backing is assuming that "a given bug was introduced by the lines of code that were modified to fix it" [52]. This assumption is made because while faulty states aren't naively visible, the commits that aim to fix them usually are, for example through commit messages or tags. This allows us to study projects from a historical perspective, vastly increasing the available data.

2.2.2 Software Risk Management

The aim of software risk management is for an organization to have a framework implemented that covers work related to the risk associated with operations [7]. The goal is to continuously identify, analyze, and assess the risk associated with a software project in a systematic, procedural manner. This can in many cases help mitigate the risk of adverse events and prepare responses for a situation in which one or several of the identified risks have occurred. There has been a substantial amount of research committed to unifying and standardizing the risk management work, both in the scope of software projects, and in more general terms. One study that is often referenced is the NIST SP 800-39 Risk Management Process [54]. This states that there are four core activities that need to take place in an organization's risk management process for it to be sufficient. These activities and the information flows necessary are outlined in figure 2.5.

- **Framing risk**, this is the step where the context of the risk is defined, and where the potential risk factors are identified.
- **Assessing risk**, this is the step where the probability and potential impact of a risk factor is examined.
- **Responding to risk**, this is the step where it is decided how to react in case of an adverse event, and what to make of the assessed risk factors. Identifying risk has no inherent value, mitigating negative value imposed through response plans and risk avoidance is what results in value.
- **Monitoring risk**, this is the process of monitoring the implementation, and effectiveness of the risk management plan decided upon through the other steps. This step is the continuous aspect of risk management and is responsible for spotting new risks, more efficient solutions and tracking.

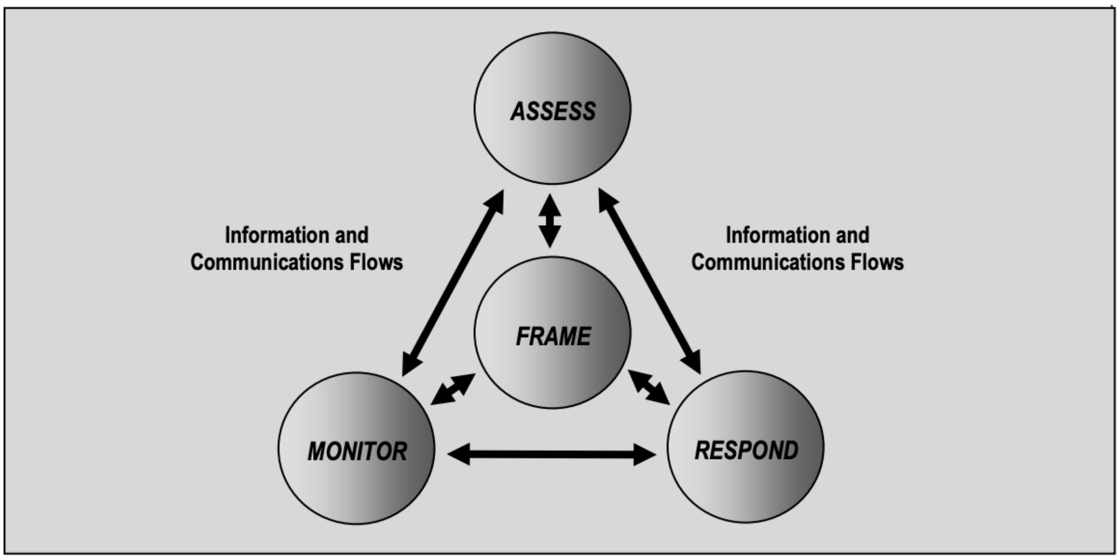

ASSESS
Information and Communications Flows
Information and Communications Flows
FRAME
MONITOR
RESPOND

Figure 2.5: Illustrating the main areas of risk management activities [54]

Software Risk Assessment

Of special interest is the risk assessment step. Risk assessment is an essential part of any software development process that involves external parties, such as any production-grade project. There are many differing definitions, and the codifying process is ongoing. There are, however, some general aspects that are agreed upon. At its most basic, it is the process of gathering information related to the process or processes of interest, in order to assess the risks related to them [54]. This often includes estimating both the likelihood that an event that would affect the project negatively would occur, and estimating the extent to which the adversity would impact the project [7]. It is a complex process that requires analyzing relevant data, both regarding the project specifically, and regarding the organization and potential end users. However, lack of frameworks and quantifiable standards leads to this being a process where much is completely dependant on human factors, mainly developer and reviewer experience, something that might limit the efficiency of the process [8]. In many organizations, the key factor in risk assessment is the collective gut feeling of the more senior competence involved. This is often a functional model in the sense that human intuition is a lean, and sometimes irreplaceable way of risk estimation. However, the approach comes with some inherent flaws, such as human biases, and constraints in the cases where the required competence is unavailable. Tying the final verdict to a select group of cross-functional, highly competent individuals can also lead to an unnecessary amount of key individuals being tied up, and to the project stagnating when not enough subject matter experts can be procured.

2.2.3 Risk Models

The results of the NIST [54] study are rather abstract and non-trivial to implement. For these reasons, it was found that few organizations actually implement the model [35], or for that matter, any clearly defined risk management model whatsoever. Many studies have been done with the purpose of concretizing a risk management workflow to the point where its readily implementable. The results of these studies are often presented in the form of a risk model. This model usually depicts a process flow, consisting of a number of activities. What activities are included, and how they are supposed to be performed vary, and in our research, a clear, de facto standard has not been identified. In this section, some of these risk models will be presented and analyzed. They can get rather complex, and explaining them all in full is outside the scope of this paper. Information that is deemed relevant for the case study will be prioritized and highlighted.

In general, it can be said that risk models offer a systematic approach to summarize and assess the identified risk factors, in terms of their relations and potential impact on a specific project. Risk factors are the potential causes of fault or failure that have been discovered. A good risk model is one that, given that the risk factors are assessed and estimated correctly, gives a valid estimation and prioritization of the risks and their respective impacts on a project. They are also an important part of representing and communicating which areas that are of extra interest in the continuous risk mitigation work [54].

By having a well defined and integrated risk model at the center of the organizations' risk management work, a higher grade of reproducibility and repeatability in the assessment can be achieved. Having a repeatable approach is also believed to lead to faster and more exact assessment due to increase comfort in the model. Having a defined model also lends itself well for a CIP, or the continuous improvement process mentioned in section 2.1.2. If the process is designed from scratch specifically for each project it is hard to utilize experience from previous projects. For this to work, however, it is important that this continuous improvement is actively worked on. The circumstances, both internal and external of a project are under constant flux, and as such, the period of usefulness of any static risk model is limited. [54]

SRAM

This model, appropriately called the Software Risk Assessment Model (SRAM) [20], was one of the first efforts to provide a modeled flow for the risk assessment process. It identifies 9 critical risk elements. These are:

1. Complexity of Software
2. Staff Involved in the Project
3. Targeted Reliability
4. Product Requirements
5. Method of Estimation
6. Method of Monitoring
7. Development Process Adopted
8. Usability of Software
9. Tools used for Development.

The model is based on a questionnaire, where a set of questions are formulated for each risk element. Each question can in turn be answered in one of three ways, in an increasing order of risk, such that a risk score between 1-3 can be given for that particular question. A normalized risk score is then calculated for each of these elements, by accumulating the numerical ratings and dividing it by the number of questions given for a particular element. This normalized value is called the risk element probability and shows what element carries the most risk, making it easier for decision makers to know where to allocate resources. The different elements can however have a varying degree of impact on a project. Thus, in order to calculate the overall risk of a project, the risk element probability needs to be weighted according to their potential impact. If we let *r1, r2, r3 ... r9* denote the nine risk probability elements, and *w1, w2, w3 ... w9* their attributed weights. Then the overall risk score for the project, *R*, can be determined by equation 2.1:

$$R = r1 * w1 + r2 * w2 + r3 * w3... + r9 * w9 \qquad (2.1)$$

Since the max risk value that can be given for each question is 3, and the minimum 1, *Rmax* and *Rmin* are obtained by equations 2.2 and 2.3:

$$Rmax = 3(w1 + w2 + w3.. + w9) \tag{2.2}$$

$$Rmin = 1(w1 + w2 + w3.. + w9) \tag{2.3}$$

A normalized overall risk score, *Rn*, that is a fraction between 0 and 1 can then be calculated by 2.4:

$$Rn = (R - Rmin)/(Rmax - Rmin) \tag{2.4}$$

In the paper, calibration of the model was then done by calculating the risk score of ten similar projects in a multinational company, and comparing them to the customer feedback index of the projects. The customer feedback index was plotted against the corresponding project risk value and linear regression was applied, as illustrated in Figure 2.6. The resulting regression line could then be used to predict the customer feedback index from a calculated project risk value. Figure 2.7 presents the predicted values against the real values.

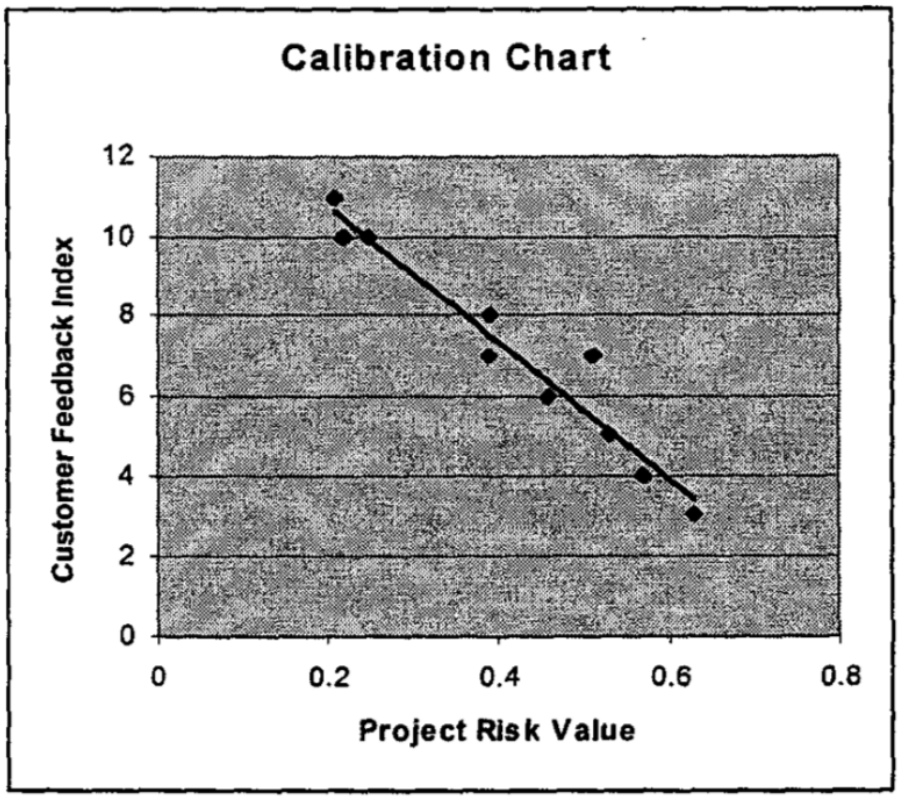
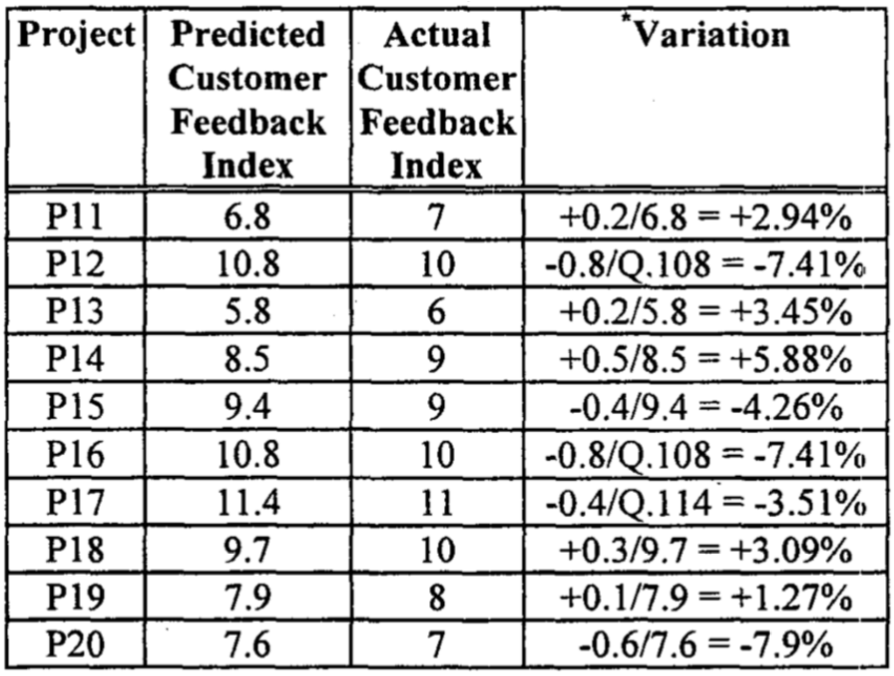


Figure 2.6: Calibration chart used in the SRAM study [20]



Project	Predicted Customer Feedback Index	Actual Customer Feedback Index	*Variation
P11	6.8	7	+0.2/6.8 = +2.94%
P12	10.8	10	-0.8/Q.108 = -7.41%
P13	5.8	6	+0.2/5.8 = +3.45%
P14	8.5	9	+0.5/8.5 = +5.88%
P15	9.4	9	-0.4/9.4 = -4.26%
P16	10.8	10	-0.8/Q.108 = -7.41%
P17	11.4	11	-0.4/Q.114 = -3.51%
P18	9.7	10	+0.3/9.7 = +3.09%
P19	7.9	8	+0.1/7.9 = +1.27%
P20	7.6	7	-0.6/7.6 = -7.9%

Figure 2.7: Illustrates the real vs predicted customer feedback values for the 10 projects included in the SRAM study [20]

The process of aggregating a weighted risk score as an end product of the risk analysis is a finding that will be considered further in the case study.

SRAEM

Another model called the Software Risk Assessment and Estimation Model (SRAEM) [23] focuses both on assessing risk as well as estimating it. An illustration of the procedure is shown in Figure 2.8. The model assumes three key risk assessment dimensions, namely risk identification, risk analysis and risk prioritization. The risk identification process aims to identify and list the risk items for a specific project. Common methods for this are: examination of decision drivers, assumption analysis, and checklists. The purpose of risk analysis is to assess the identified risks, in terms of their probability of occurrence and the magnitude of their impact. The commonly used techniques for this include performance models, cost models, and network analysis. Finally, the risk prioritization procedure should result in a ranked list of the risk items that have been identified and analyzed. The typical ways of doing this include risk exposure analysis and risk reduction leverage.

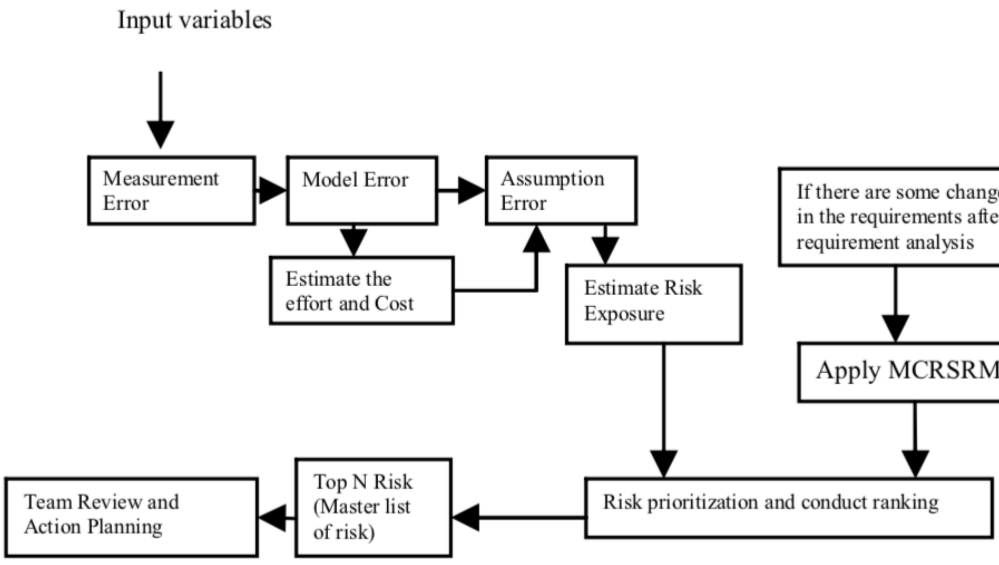

Input variables
Measurement Error
Model Error
Assumption Error
Estimate the effort and Cost
Estimate Risk Exposure
If there are some changes in the requirements after requirement analysis
Apply MCRSRM
Risk prioritization and conduct ranking
Top N Risk (Master list of risk)
Team Review and Action Planning

Figure 2.8: The Software Risk Assessment and Estimation Model (SRAEM) [23]

The initial steps of the model are to estimate the sources of uncertainty: measurement error, model error, and assumption error. These are then considered when assessing the risk, in which probabilities and magnitude of impact are estimated. The model then proposes two ways of performing the risk prioritization. The first one is through risk exposure, which is done by multiplying the probability of a risk occurring with the impact it would have. That is, if there is a 10% probability of a risk occurring, and this would have a cost of 1000, the risk exposure would be *0.1*1000 = 100*. The risk exposure values can then be compared, making it possible to rank them. Another method to rank the risk is to utilize so-called mission critical requirements stability risk metrics (MCRSRM). If one or several requirements have been changed, the total risk of this can be calculated and given a value. If we let R denote the risk added from changing requirements, then the added risk can be calculated according to equation 2.5.

$$R = b/a + K(A(c/d) + B(d/b) + G(e/b)) \tag{2.5}$$

Where a = total number of requirements, b = total number of Mission Critical Requirements (MCR), c = number of MCR added during phase, d = number of MCR modified during the phase,

e = number of MCR deleted during the phase and K, A, B and G are the penalties for adding, modifying or deleting of requirements.

Ultimately, this procedure should present a list of identified risks and their corresponding risk indexes. This can then serve as a basis for ranking and prioritizing the different risks and decide on an appropriate contingency plan.

CORAS

CORAS is a european framework for model-driven risk analysis [39]. The risk assessment model is constituted of five steps.

- Hazard and operability study.
- Fault tree analysis.
- Failure mode and effect criticality analysis.
- Markov analysis methods.
- CCTA Risk Analysis and management methodology.

Of extra interest is the fault tree analysis (FTA), shown in Figure 2.9. It is a risk assessment method. With enough probability data, it would be possible to construct a quantifiable model, something that would lend itself well for automation. FTA is a method that is based on a general system-safety analysis technique. Its steps can be extended to be useful even in a software context. This modified and extended approach is known as the software fault tree analysis (SFTA). Undesired events are defined as software failures and faults. STFA is used to examine which adverse events that are most likely, and also which underlying events that might have caused these top-level failures.

Failure event ordering is done in a tree shape, with the predicted software failure as a parent root. The underlying causes that lead to these faults are then children nodes. Basic software failure events and events that too little is known about form the leaves of the tree. The top root node of the tree often represents catastrophic or at least total failure.

Ordering the fault events and their relation to the underlying causes this way has some benefits. It can be used to:

- Understand what causes lead to a certain faulty state.
- Demonstrate that it is impossible for a system to reach catastrophic failure
- Calculate aggregated risk of the system entering a certain faulty state, given the contributing factors

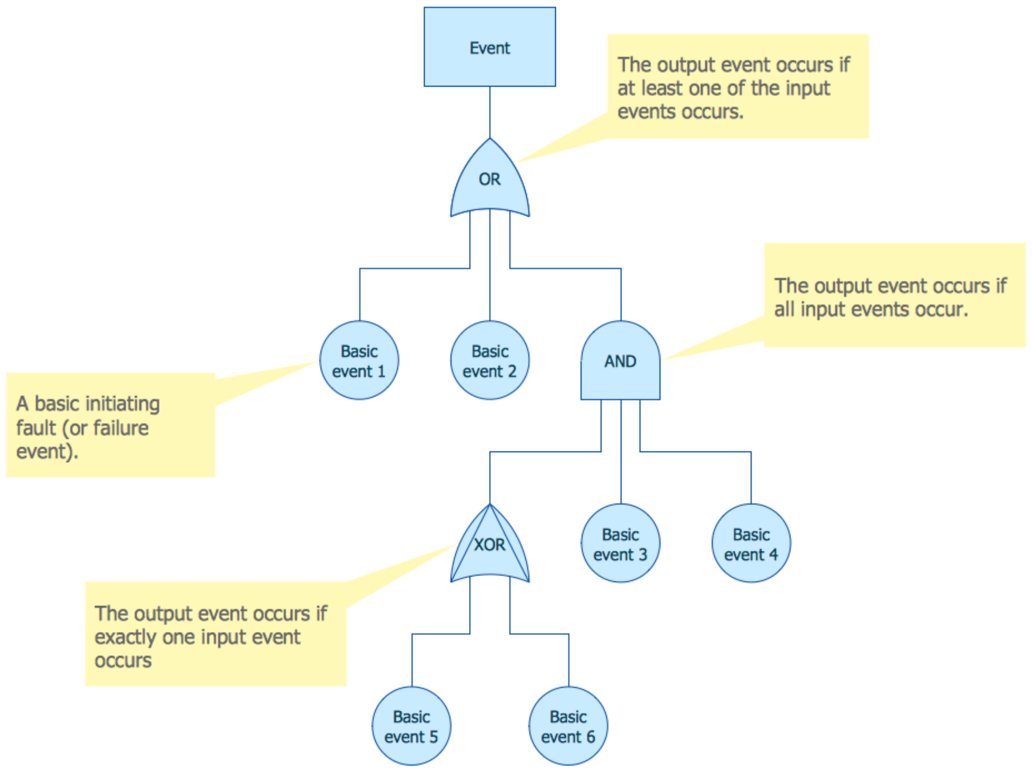

Event
The output event occurs if at least one of the input events occurs.
OR
Basic event 1
Basic event 2
AND
The output event occurs if all input events occur.
A basic initiating fault (or failure event).
XOR
Basic event 3
Basic event 4
The output event occurs if exactly one input event occurs
Basic event 5
Basic event 6

Figure 2.9: Example of software fault tree analysis applied on airport security [62]

SRAEP

SRAEP, or Software Risk Assessment and Evaluation Process [56] is a risk assessment model that is an extension of the SFT (Software Fault Tree) method to identify risks. The model states three dimensions of software fault: Technical Risk, Organizational Risk, and Environmental Risk. The technical risk stems from not having the necessary know-how to perform a task, or having a poorly defined task to begin with. The organizational aspect is a result of lacking internal communication and responsibility assignment. Environmental risk arises as a consequence of changing environments and issues in external communication. An illustration of the process can be seen in Figure 2.10.

This model is based on five activities.

- Identify context
- Identify risk using the software fault tree
- Compute risk exposure
- Risk prioritization
- Action planning

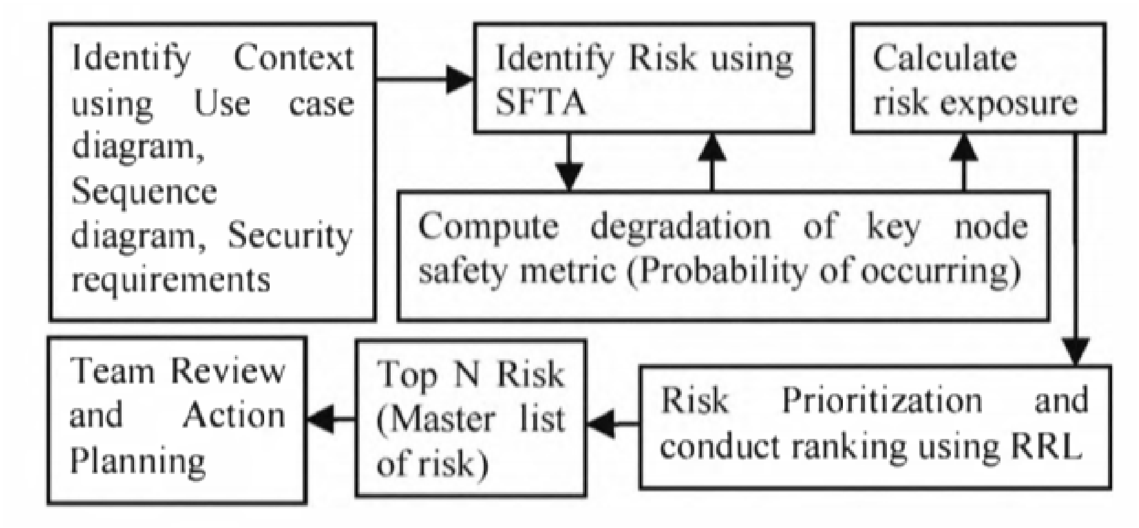

Identify Context using Use case diagram, Sequence diagram, Security requirements
Identify Risk using SFTA
Calculate risk exposure
Compute degradation of key node safety metric (Probability of occurring)
Team Review and Action Planning
Top N Risk (Master list of risk)
Risk Prioritization and conduct ranking using RRL

Figure 2.10: Software Risk Assessment and Evaluation Process (SRAEP) [56]

The interesting part of this model is its quantifiable way of calculating RRL, the risk reduction leverage, or cost of countermeasure. This is a way of assigning a numerical value to the impact a certain adverse event would have compared to the cost of mitigating the risk. At some point the expected cost of reducing risk further might be higher than the expected cost of not mitigating the risk. This metric is calculated according to equation 2.6, where *RRE* is the reduction in risk exposure and *C* is the cost of counter measure.

$$RRL = RRE/C \tag{2.6}$$

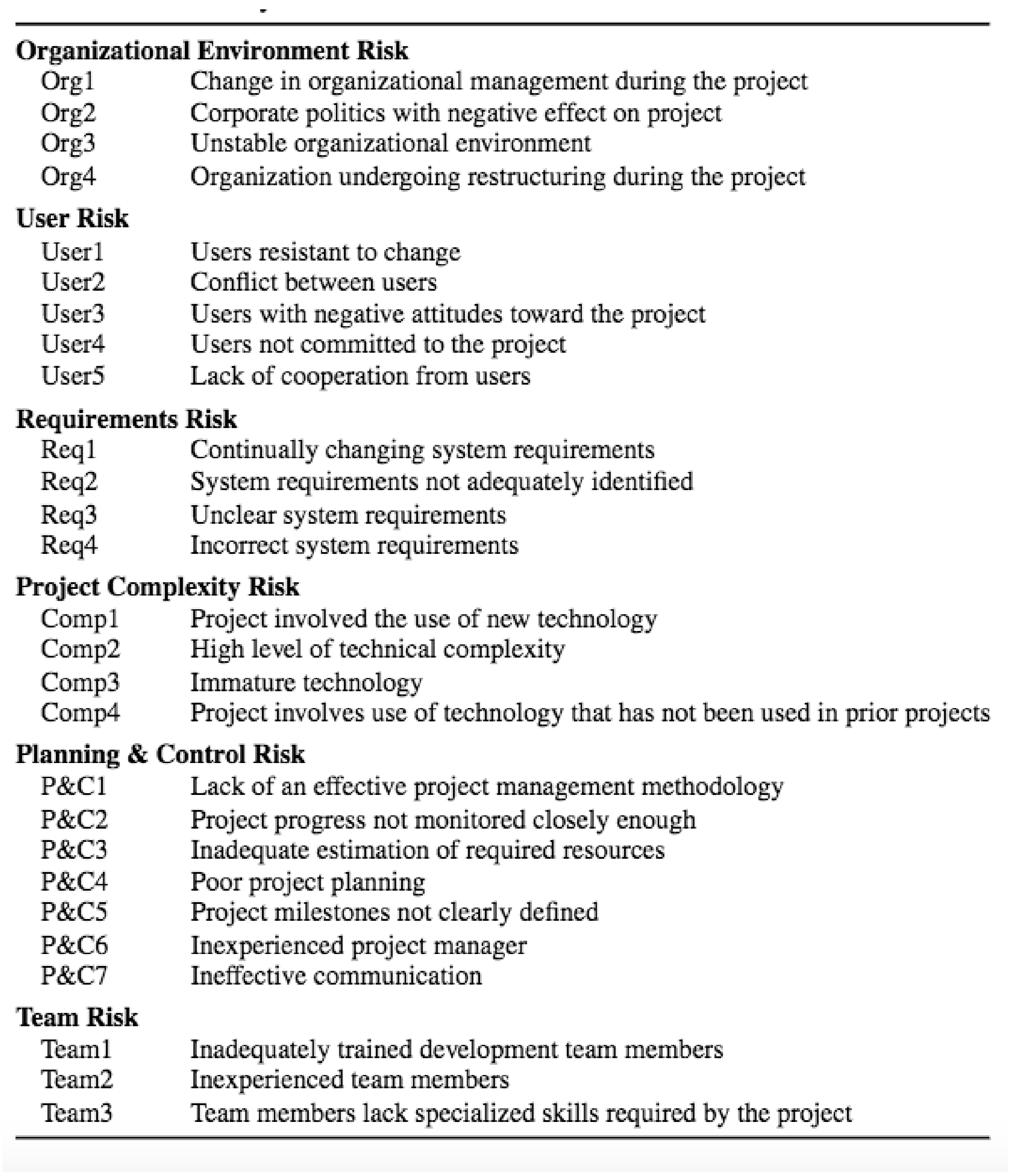
2.2.4 Risk Factors

Identifying the factors that induce risk in a software development project is critical for the risk assessment process, and a first step in many of the models presented above. There can be a high number of these factors, and the amount of risk they introduce varies heavily [36]. A challenge when designing a risk assessment tool or model is therefore to identify a set of risk factors that provide an adequate prediction of the risk, without making the process overly complex [54]. Some, but not all, of the models have taken this process into account. The following section aims to further investigate which factors are the most important to consider. The consistency of how often a factor shows up in between studies will be considered a signifier of importance, as well as the soundness of the theoretical backing. To complement these found factors, there will also be a considerable amount of hypothesizing. Factors that have not been thoroughly considered in the studies will be summarized for further scrutiny in the case study and generalization sections. A few approaches to identify risk factors have been made through survey studies. An international study by Schmidt et al. [57] identified and produced a ranked list of 29 software risk factors. The study was based on a Delphi survey methodology, where the respondents were asked to list what they thought were the most critical risk factors when developing software. The results represent the survey responses from a panel consisting of 41 members, all of whom were experienced project managers. Iranmanesh et al [34] later divided the factors from this ranked list into twelve categories, making it more general, and easier to overview. The results of these studies are relatively comprehensive and available in Appendix A.

Another, similar study, in which experienced project managers were surveyed, was performed by Boehm [7]. The resulting list of 10 risk factors has been widely used in practice and is well-known [53]. Ropponen and Lyytinen [53] later expanded on Boehm's research and conducted a comprehensive survey study of their own. In total, they surveyed 83 project managers with combined experiences of nearly 1100 projects. They mention that Boehm's list lacks theoretical background as the list of risk factors has been compiled inductively, and that some factors therefore refer to the same thing. Rapponen's and Lyytinen's revised list groups these factors together into one independent software risk component, which makes the different factors more distinct from each other.

Furthermore, various other attempts have also been made to identify common risk factors. Moynihan [42] conducted interviews with 14 experienced application developers. Literature studies by Barki [4] and Thakur and Singh [61] will also be considered in this paper. All factors that have been presented in the studies are listed in Appendix A.

By looking at the identified factors in the different studies, it is apparent that they are described in a variying degree of detail and level of abstraction. Many are also referring to the same root cause, although described differently. This was the reason for Rapponen and Lyytinen's, as well as Iranmanesh's, more general and distinct categorization of the factors. Rapponen and Lyytinen mention that "this would provide a simpler basis for developing instruments to measure software risk" [53]. A comprehensive literature review by Wallace et al [64] concludes a model with 6 risk dimensions and their corresponding main risk factors. The study includes the papers by Schmidt and Lyytinen [57], Barki [4], Boehm [7] and Moynihan [42]. Because of the comprehensiveness and the distinct categorisation of risk, this model will serve as the basis for what factors will be considered in the risk assessment. The table of risk factors that Wallace presents can be seen in Figure 2.11.



Organizational Environment Risk	
Org1	Change in organizational management during the project
Org2	Corporate politics with negative effect on project
Org3	Unstable organizational environment
Org4	Organization undergoing restructuring during the project
User Risk	
User1	Users resistant to change
User2	Conflict between users
User3	Users with negative attitudes toward the project
User4	Users not committed to the project
User5	Lack of cooperation from users
Requirements Risk	
Req1	Continually changing system requirements
Req2	System requirements not adequately identified
Req3	Unclear system requirements
Req4	Incorrect system requirements
Project Complexity Risk	
Comp1	Project involved the use of new technology
Comp2	High level of technical complexity
Comp3	Immature technology
Comp4	Project involves use of technology that has not been used in prior projects
Planning & Control Risk	
P&C1	Lack of an effective project management methodology
P&C2	Project progress not monitored closely enough
P&C3	Inadequate estimation of required resources
P&C4	Poor project planning
P&C5	Project milestones not clearly defined
P&C6	Inexperienced project manager
P&C7	Ineffective communication
Team Risk	
Team1	Inadequately trained development team members
Team2	Inexperienced team members
Team3	Team members lack specialized skills required by the project

Figure 2.11: Wallace's table of risk factors, divided into 6 risk dimensions.

2.2.5 Risk Representation

The results of a risk assessment or analysis through the use of a risk model or systematic approach only rarely provides a fully formed action plan [16]. Rather, their function is to provide a basis for discussion and further response planning. To allow for this it is important that the risks and impacts are clearly communicated. Systems can be complex, and statistics is a field that might not be completely intuitive, especially for stakeholders without mathematical background. This is where risk representation is important. Risk representation concerns how to best represent the results of a study in a way that lends itself to value creation in terms of adverse events mitigated.

The Institute of Risk Management (IRM) recommends quantifying probability using the following scale: high (more than 25%), medium (between 25% and 2%), and low (less than 2%) [65].

Color Coding

One hypothesized way to represent this could be through the use of color. The risk factors, and their contribution could potentially be separated, and then color-coded depending on severity. One example might be green/yellow/red, where green represents that less than 1% of commits with this factor value caused a faulty state. Yellow meaning that more than 1% but less than 10% of the commits have caused a faulty state. Red means that over 10% of commits with this value has caused a faulty state. Example, we look at three hypothetical commits. One with 1500 insertions, one with 130 insertions, and one with 4 insertions. If we find significant factors it is likely that the three commits would be placed in category red, yellow, green respectively. This is an easily glanceable and project-independent way of representing the risk contribution of the various factors.

Radar Graph

Another way of representing the risk factors could be in a radar graph, as shown in figure 2.12.

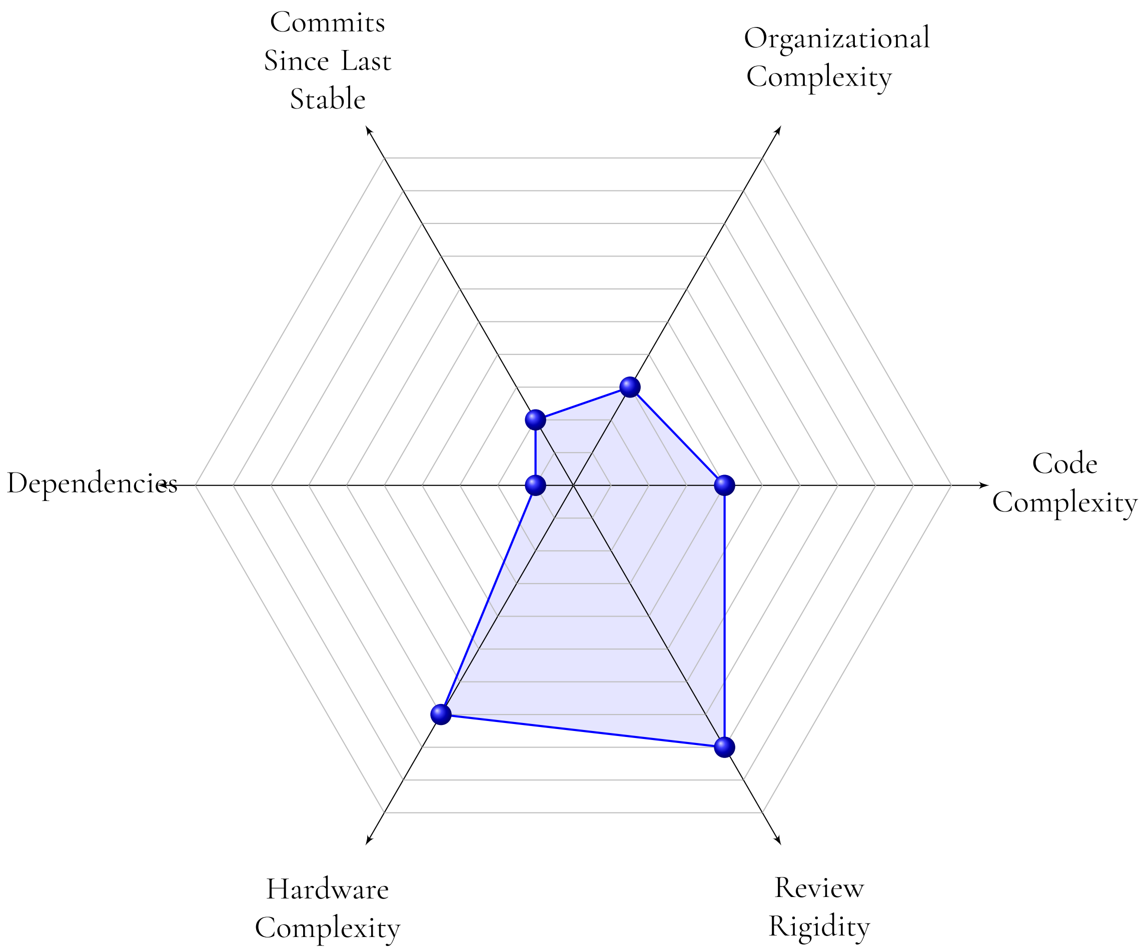


Figure 2.12: Risk factor representation through radar graph

2.3 Code Quality Metrics

When code is in a functionally faulty state, this is possible to spot through errors and erroneous behavior of the program. However, two pieces of code that both perform the same task perfectly well can still be in a more or less healthy state [31]. One can be built with classes cross-referencing, variable names being re-purposed and a general lack of cohesion, whilst the other follows all the best practices, leading to a more valuable piece of software. This is often referred to as software structural quality [13] and the difference is often impossible to spot without a static review of the actual codebase. This code health is often measured through something that in this paper will be referred to as Code Quality Metrics, these metrics aim to study the internal integrity of a codebase. The four core factors that are to be studied in this report are coupling, complexity, cohesion and size, and it is hypothesized that these metrics factor in to the risk assessment process [12]. A change affecting a very complex element might show a higher risk of causing adverse effects. A change affecting a very big module might have a higher impact in case of an adverse event.

2.3.1 Coupling

Coupling is a measurement of module interdependence. This dependence can range from very high (also "tight" or "strong"), as is the case when modules share code, to very low (also "loose" or "weak") which is when modules are heavily self-reliant and contained [31].

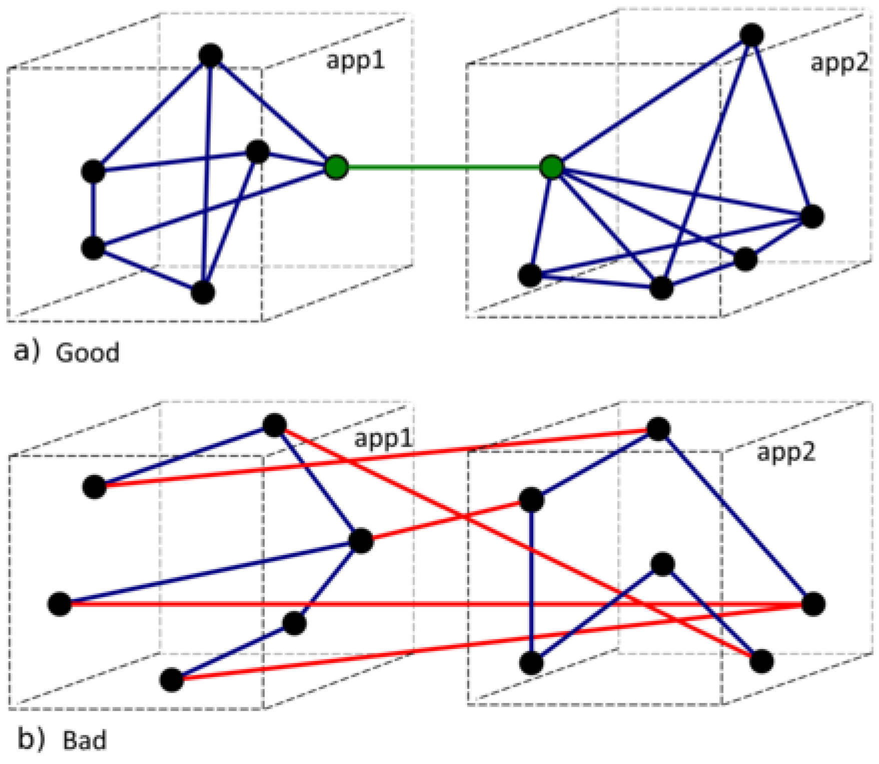


Figure 2.13: Two modules, one with light coupling, one with heavy coupling [63]

Schema a) shows two modules that are loosely coupled. They have connections inside the module but the outward facing communication paths are limited. Schema b) shows two modules that are heavily coupled.

Low coupling is considered desirable, whereas high coupling can lead to a host of adverse side-effects in the maintainability and readability scope. Some examples of problems that are more likely in highly coupled systems include [66]:

- Changes in one file can have a ripple effect that either affects or demands changes in other modules. This affects both the risk induced and the potential impact of a change [9].
- Module assembly is harder when having to take multiple interdependencies into account.
- Modules might be harder to test and reuse since they exist in a context rather than a vacuum.

The hypothesis is that a change affecting a highly coupled class has both a higher risk of an adverse event, and higher impact when something goes awry. Changing a highly coupled element demands knowledge about the elements it is tied to.

2.3.2 Cohesion

Cohesion is an ordinal measurement of how strongly the elements in a module belong together [63]. In contrast to coupling, high cohesion is generally considered favourable.

Coupling is often measured on a scale that goes from two extreme cases.

- the components are packaged according to interconnections, and the interfaces that interact with elements outside of the package are limited. If all of the elements of the module also contribute to a singularly defined task this is called functional coherence, and is commonly deemed best practice [63].
- is a a much less structured approach where the only connection between elements seems to be the fact that they have been put in the same package. This is referred to as coincidental coherence, and offers no tangible benefits to the codebase as a whole [5].

High cohesion is said to lead to a couple of benefits [33]:

- Lower individual module complexity
- Increased maintainability as a consequence of increased module atomicity
- Increased module reusability, especially in the case of functional cohesion

The hypothesis is that a lack of cohesion increases the risk of adverse events mostly due to lack of readability. An adverse event might also result in a bigger impact since one failure in one part of a module that performs several functions can lead to several functionalities being unavailable rather than just one, as would have been the case if the module was functionally coherent.

2.3.3 Complexity

Complexity is a measurement of how hard the module is to understand, and a general grade of abstraction and algorithmic loftiness [12]. Interactions that involve a large number of sub-processes and elements also tend to warrant a higher complexity score. A high level of complexity increases the risk that a change to the module would have been made without sufficient understanding. As such, there is a bigger risk that the change causes adverse effects, and also that these effects can be harder to spot and remedy, leading to a greater impact.

2.3.4 Size

Perhaps the most intuitively graspable measure so far. The size of the code-base that constitutes a module is thought to be connected to maintainability and robustness. Both length (lines of code) and number of functions are metrics that can be used to measure the software size [17]. Too great a size might allude to that the module is doing too much, and too low might point to that the arrangement is unnecessarily complex. Furthermore, it is possible that changes that affect a large code ecosystem have a generally greater impact on the program in the case of adverse events.

2.3.5 Code Quality Metrics in the Risk Assessment Process

A majority of reports on the subject show that structural functionality is a key aspect to consider in order to fulfill the non-functional requirements often related to an artifact [33]. Maintainablility, readability, and stability tie directly into the risk of adverse events and there is therefore theoretical backing that this area is relevant in the risk assessment process. However, there are some aspects that needs to be considered in the context of our report. Assessing these metrics manually is a time-consuming, and not very generalizable undertaking. As such, if the goal is to involve them in an automated and DevOps compatible pipeline according to the best practices mentioned in section 2.1, there needs to be tools that assign cardinal values to the different areas of interest. Another prerequisite for this to be feasible is that this tool has an API that can provide these values in a machine-readable format, so that the step can be integrated into the pipeline. This will be further explored in the case study.

2.4 Machine Learning

One area that will be explored during the case study is whether machine learning can be used to study the connection between extracted risk factors and bugs induced. Machine learning is an area of science that has increased vastly in prevalence during the last decade [45]. At its most basic, it can be defined as a systematic and procedural approach to determining statistical connections between process' inputs and outputs, for which it has not been specifically programmed [55]. In the scope of this project, the inputs would be the extracted factors and the output would be the observed adverse results of a commit. Sensitivity analysis would also potentially allow us to study the individual importance of factors, thus arriving at a weight that could be used to increase traceability of the service's results.

2.4.1 Overfitting

One of the most common problems when applying machine learning is that the model overfits the data [2]. Such is the case when the model gets too sensitive to the training data, and rather than learning a generalizable correlation, it learns the specific data set. Such a case is illustrated in figure 2.14. It is highly likely that distribution of datapoints exist on the straight line, but the model has found a connection that allows for all the datapoints of the training data to be fitted exactly. This in turn, would lead to highly volatile and misrepresenting predictions.

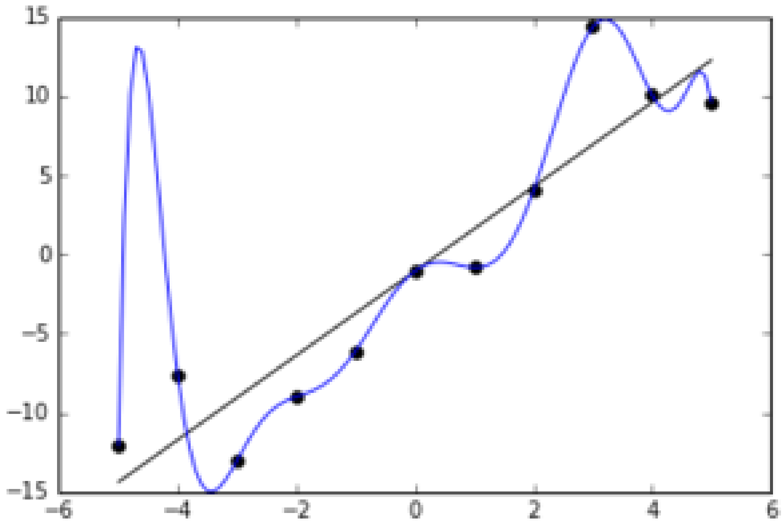


Figure 2.14: Example of an overfitted model [1]

There are several ways of avoiding this. Some models avoid overfitting automatically. It is also possible to tune hyperparameters, such as regularization applied, or wrangle the data, such as through bootstrapping or sampling. A model's predictive performance on data it has been trained on will always converge towards perfect, given enough time and network depth. In order to avoid these falsely optimistic results it is essential to have a test data-set, which the model has not been fed during the training process [2]. The idea is that this data simulates the model being tried on hitherto unseen data, as is the real-world use case most of the time. One common way of achieving this is simply splitting the data-set into two chunks, one large one that the model is trained on, and one smaller one that the will be fed to the already trained model. The model's performance on this smaller, unseen, dataset is the performance metric that is thought to be the most representative on how it would perform in real-life applications.

2.4.2 Performance Metrics

In order to fruitfully gauge a trained model's performance, it is important to define metrics that accurately represent how well it is performing certain tasks. In this section, a few selected metrics will be listed. The metrics in question are based on a paper on performance metrics for software fault prediction [11]. Some terms that form the basis for the metrics are the following:

- Positive and Negative. These are the terms that are used to signify the two states possible in binary classification. In our case, a positive instance is something that induces a bug, whereas negative case does not.

- False and True. These concern the model's guess in relation to the actually observed state. True signifies a correct labeling, false an incorrect one.

From this we can infer that there are four different states that can follow a binary classification, as shown in Figure 2.15 below:

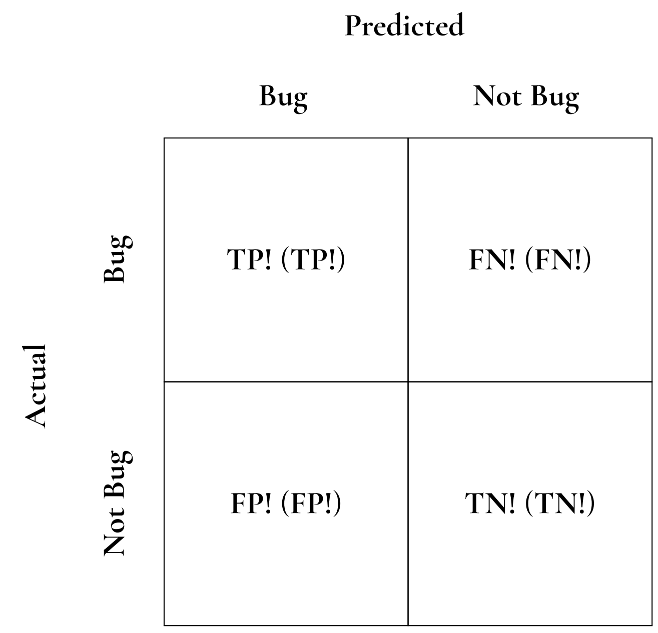


Figure 2.15: Illustrates the meaning of true positives, true negatives, false positives and false negatives.

Sensitivity and Specificity

Sensitivity and specificity are perhaps the most intuitively graspable performance metrics of a binary classifier. Sensitivity is a measurement for how great the model's capability of catching positive events are. The metric for this is quite simply predicted positive cases over observed positive cases *(TP/(TP+FN))*. Specificity is a measurement for the model's capability of disregarding (not flagging) negative cases. In analogous fashion to sensitivity, this is measured by dividing predicted false cases with observed false cases *(TF/(TF+FP))*. These can be treated as separate entities, since it is, in theory, possible for both metrics to achieve 100%. However, in practical application, there's usually a trade-off where tuning the threshold can increase sensitivity, but lowers specificity and vice-versa. The intuitive approach to this relation is thinking in terms of stringency. A model that is instructed to rather be safe than sorry is gonna catch slightly more of the positive events, but as an effect might have an increased number of false positives. There are cases where this trade-off is worth it. This relationship between sensitivity and specificity is usually illustrated with a Receiver Operator Characteristic (or ROC) curve, explained below [44].

ROC-curve

Since threshold cutoff can be tuned to increase the sensitivity, but decrease the specificity and vice versa, it can be inferred that these have a mathematical correlation. For models that take other things in consideration than just making the maximum amount of right predictions (such as models that verge on the "safe" side, or where the fuzzy logic prediction score is the main takeaway), there needs to be a way of illustrating this cutoff in order to understand just how much specificity has to be offered for extra sensitivity. This relationship is illustrated with the ROC-curve [22]. This is a 2d-graph with true positives on the y-axis, and false positives on the x-axis. Figure, 2.16 illustrates some fairly typical ROC-curves. A high true positive rate and a low false-positive rate is desirable. For a perfect classifier, the graph would be hugging the left and top walls.

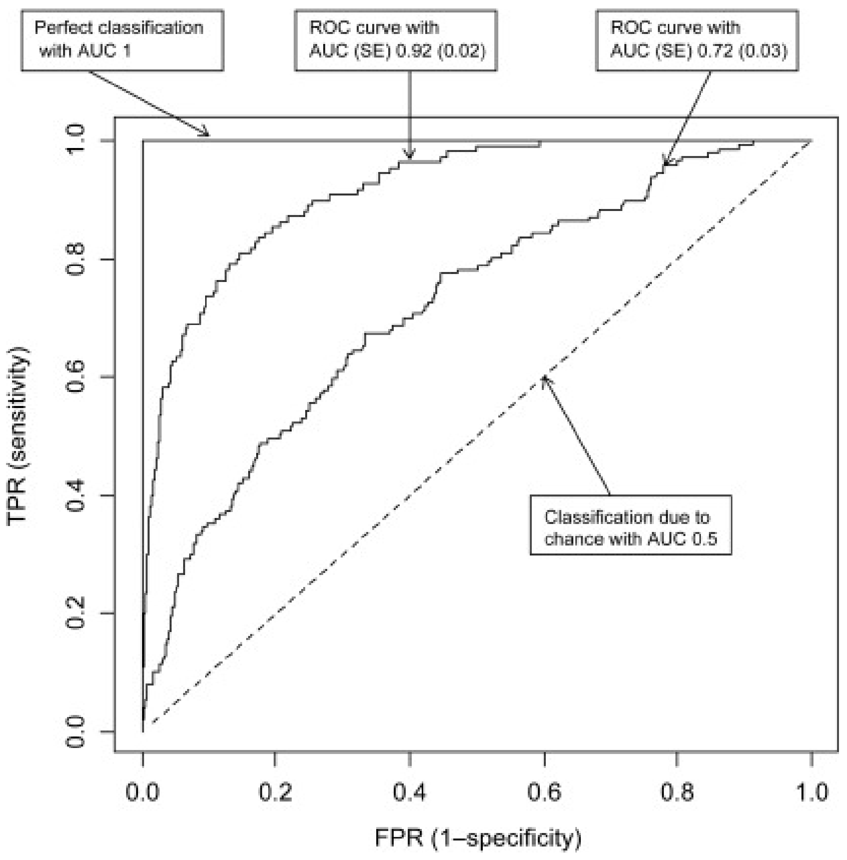


Figure 2.16: ROC-curve illustrated [47]

Another metric that follows the ROC-curve is the AUC, or area under curve [22]. This is quite simply the area under the ROC-graph of the model. For models where one wants to be able to fine-tune the sensitivity, this value is thought to be a more representative metric of the model's performance than flat precision. A perfect model would have an AUC-score of 1.0 whereas one that predicts completely at random would have 0.5 [22]. It is important to note that this is true even for unbalanced datasets. An approximate classification of what the different AUC-scores mean is presented below [60]:

- .90-1 = excellent (A)
- .80-.90 = good (B)
- .70-.80 = fair (C)
- .60-.70 = poor (D)
- .50-.60 = fail (F)

However, how good an AUC-score is varies from use-case to use-case [10]. If one were to device a model that guessed equity returns with an AUC of 0.7, huge profits could be made from it, whereas an AUC-score of anything below 0.95 for i.e. handwritten digit classification would be seen as sub-par by today's standards. Generally, the less determinate your data is, the harder it is to reach a high AUC-score.

Chapter 3

Case Study

To assert whether our research findings could help in day-to-day software development practice, a case study was carried out at the company Sinch. Here, the goal was to study what factors have historically had the most impact on whether a commit or deployment failed or not, and also if the existing reviewing and change acceptance process could be scaled down in favor of a more automated approach.

The case study was performed in three main steps. Firstly, the company's current situation was assessed, especially in regards to the context in which the risk assessment process takes place. This was done through perusing related documentation and through interviews with stakeholders in various roles connected to the process. The second step was to combine the theory from Chapter 2 with the findings from the applied study in order to scope the risk assessment and design a service that fits Sinch AB's needs. Part three consisted of an evaluation and validation of the data procured, and the quality of potential risk assessment.

In order to answer the research questions, there were two main stages of development that are required. First, in order to answer question R1, whether there are risk factors that can be extracted and harnessed to assess risk, we needed a way of extracting values of potential risk factors, finding the historical adverse events, and connecting the two. The amount of data that needed to be studied to confirm suspected correlation necessitated an automatable extraction and labeling process. Furthermore, in order to answer question R2 we needed to design a service that lends itself well for usage in a DevOps pipeline, and that actually realizes business value for the company. The implications and limitations of the implementation of such a tool will be discussed in the following sections. A design will be proposed based on the results from interviews with stakeholders, theoretical research, and what was possible to implement within the time frame of the project. The interdisciplinary nature of the scope opens many doors to really dive deep into any one specific area, but our primary goal was to answer the research questions in full.

Sinch

Sinch is a medium-sized software company based in Malmö, Sweden. Its core service lies in providing API:s that bridge the gap between internet and the GSM networks. This allows companies to automate and facilitate their customer contact and engagement, mainly through SMS services. Due to a couple of reasons, two-factor validation being a major one, the company has enjoyed rapid growth during the last couple of years. With rapid growth comes problems of implementing and scaling processes and standards that were sufficient for a smaller business, but that might be hard to adhere to in a bigger, more complex organization. One of these procedures that has been lagging behind is the risk assessment process. According to people interviewed at the company, the process has been left largely untouched since it was first designed. The company is now trying to become ISO-27001 certified, and a large step in this direction is having an automated last-line-of-defence deployment risk assessment.

3.1 Current Risk Assessment Process

The end goal of this thesis is to facilitate Sinch's risk assessment process. In order to do this, it is essential to understand what the process looks like currently, and what functionality is demanded of it. This is what will be studied in this section. Work pertaining to the risk assessment is done at all stages of development and involves multiple stakeholders. As such, a broad perspective is taken at first. Later, areas that are deemed of extra interest or importance will be honed in on. In order to get a broad view, interviews were conducted with stakeholders in a variety of roles as to get an as multifaceted view of the risk management process as possible. We had an ongoing dialogue with Jonas Schultz, our supervisor and head of the DevOps team in Malmö (*Tech Lead Manager DevOps*). In addition to this, we conducted two formal interviews. One with Johan Brodin (*Director Self Serve Common Services*), ex. Product Owner of the MSL project and Jonas' current boss, and one with Mikael Thyman (*Tech Lead Manager*) and Thomas Forsström (*Tech Lead*), who have both been involved in the RAS-process, and have extensive experience in the MSL project. We have also performed a prototype demo that involved Johan and Jonas. During the interview with Michael and Thomas we got a better understanding of the developers' perspective of the RAS-process. We talked about potential risk factors, and aspects of the process that they think are important to take into consideration going forwards. With Johan we talked from a product owner and organizational perspective. We defined the purpose of the RAS-sheet and outlined how we design a service that people will actually use and benefit from. We also, more informally, gathered opinions and insights from people around us during the short while when there was still people at the Sinch offices. Furthermore, data from some of the larger current projects at Sinch has been provided for exploration and testing purposes.

3.1.1 Development Pipeline Layout

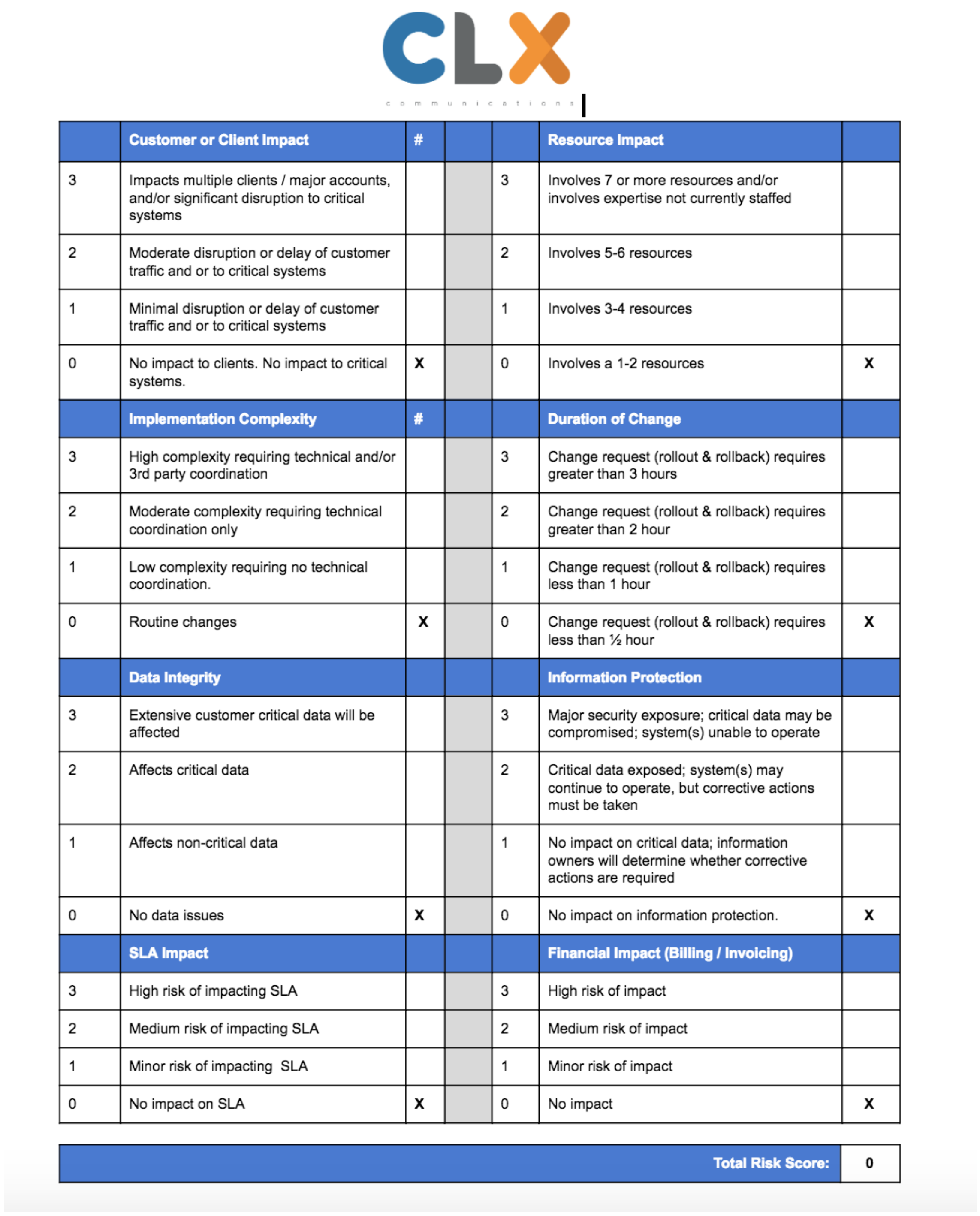
As mentioned earlier, the risk assessment process is interwoven into all stages of development and maintenance. To understand how, it is important to have a basic understanding of the development cycle that is currently employed at Sinch AB. Interviews made it clear that the approach varies slightly in between different teams, functions, and projects. There are, however, guidelines and best practices in place that are implemented often. In the project data provided, there are also projects that follow these well, which in turn afford a solid ground to base hypothetical patterns on. For the purposes of this paper, it is assumed that these practices are reasonably adhered to. While it is likely that the findings presented can be extended to other projects that don't strictly adhere to these guidelines, that will be considered a bonus rather than a requirement. Sinch AB utilizes the built in CI/CD framework in the GitLab ecosystem. This is a DevOps tool that has the capability to manage an artifact's, or change's journey through the steps "Verify" to "Release" in the DevOps pipeline outlined in figure 2.1. This pipeline consists of several stages, that all handle the element according to a predefined procedure. Each stage of the pipeline generally have the ability to cancel a change or deployment in the case that the requirements of that particular stage aren't met.

Generally, when a potential change is conceived, it is saved as a ticket in JIRA, an issuing management system. A team or developer starts working on the ticket locally, utilizing git as the main versioning tool. Continuous delivery is already implemented in most of the developer competences' workflow. In order to deliver the change to a common repository it has to pass a gauntlet that builds and tests the artifact. It is after this process that the developer or team has to assess the risk posed by deploying the change.

3.1.2 Risk Assessment and Reviewing Process

Currently, the assessment of whether a change constitutes a risk or not is solely based on a single sheet, referred to as the RAS-sheet, presented in figure 3.1. If any of the eight categories present a score of one or higher, the change has to be submitted to a change acceptance board meeting (CAB) before deployment. The change acceptance board consists of senior personnel, meaning a reduction of the review time could lead to significant cost savings, and freeing up key resources. By having a tool that generates, or at least gives a basis for further decisions, the review process would be greatly facilitated. From interviews with Sinch employees, the lack of data to support a score in the sheet is one of the most prominent experienced shortcomings. Figure 3.1 shows a case where the sheet has already been filled in (the x's). It illustrates the only change magnitude that does not warrant a CAB-meeting, as all the categories show straight zeroes. If even one category would be deemed a higher risk, the total risk score would reflect this and a meeting would have to be called. Interviews show that some projects have a rollout that takes more than half an hour. In this case, even the most menial change will score a 1 or higher in the "Duration of Change" field, which in turn will necessitate a meeting. This removes agency from the people who are actually in charge of the project and is costly in terms of man-hours.

In Figure 3.1, we can see the different risk categories, or dimensions, Sinch currently uses to assess the deployment risk. When comparing the sheet with the proposed risk dimensions in Figure 2.11 from Section 2.2.4, we can conclude that only project/implementation complexity and SLA impact/requirements risk are present in both. This is likely due to the fact that Sinch's RAS sheet focuses more on risk regarding a specific change and not the entire project, like Wallace [64] does in the proposed figure. This is likely due to Sinch having developed the RAS sheet with the DevOps workflow in mind, where deployments need to be assessed on a more granular level. Projects do not conclude in a final delivery, but services that constantly need to be maintained and updated through various deployments. The four risk dimensions in Figure 2.11 that are omitted from the RAS sheet, are factors that would remain constant, i.e. be the same for individual changes but likely vary from project to project. It is likely that taking these dimensions into account could further facilitate the risk assessment, by providing some project-level context. It is also worth noting that a vast majority of the dimensions in the RAS sheet are focused on assessing the impact of an adverse event, rather than the probability. For a full risk assessment, a combination of these two aspects are required, as discussed in our two-component definition of risk in Section 2.2.



CLX communications

	Customer or Client Impact	#			Resource Impact	
3	Impacts multiple clients / major accounts, and/or significant disruption to critical systems			3	Involves 7 or more resources and/or involves expertise not currently staffed	
2	Moderate disruption or delay of customer traffic and or to critical systems			2	Involves 5-6 resources	
1	Minimal disruption or delay of customer traffic and or to critical systems			1	Involves 3-4 resources	
0	No impact to clients. No impact to critical systems.	X		0	Involves a 1-2 resources	X
	Implementation Complexity	**#**			**Duration of Change**	
3	High complexity requiring technical and/or 3rd party coordination			3	Change request (rollout & rollback) requires greater than 3 hours	
2	Moderate complexity requiring technical coordination only			2	Change request (rollout & rollback) requires greater than 2 hour	
1	Low complexity requiring no technical coordination.			1	Change request (rollout & rollback) requires less than 1 hour	
0	Routine changes	X		0	Change request (rollout & rollback) requires less than ½ hour	X
	Data Integrity				**Information Protection**	
3	Extensive customer critical data will be affected			3	Major security exposure; critical data may be compromised; system(s) unable to operate	
2	Affects critical data			2	Critical data exposed; system(s) may continue to operate, but corrective actions must be taken	
1	Affects non-critical data			1	No impact on critical data; information owners will determine whether corrective actions are required	
0	No data issues	X		0	No impact on information protection.	X
	SLA Impact				**Financial Impact (Billing / Invoicing)**	
3	High risk of impacting SLA			3	High risk of impact	
2	Medium risk of impacting SLA			2	Medium risk of impact	
1	Minor risk of impacting SLA			1	Minor risk of impact	
0	No impact on SLA	X		0	No impact	X

Total Risk Score:	0

Figure 3.1: The Risk Assessment Sheet (RAS), currently the basis for the deployment risk assessment process at Sinch

It is believed that the chance of our service fully replacing the RAS-process is low, at least in the time frame afforded by the project. Extracting accurate data on for example Client or SLA impact, two of the fields shown in Figure 3.1 is believed to be hard, due to the soft nature of the documentation involved. Furthermore, some areas, such as Resource impact, Data Integrity, and Financial Impact are trivial for the developers to estimate correctly, rendering the potential alleviation of automation smaller. It is clear, however, that there are areas where even a rather basic result, such as data about the employment, could facilitate the process. Interviews show that there is a consensus regarding that Implementation Complexity has been particularly difficult to estimate. It seems that it is also the category most likely to cause significant adverse impact in case of a misjudgment, i.e. being the area most closely related to risk probability, an aspect in which the current assessment process is somewhat lacking. Furthermore, it is the category in which most traces are left during the development and monitoring process. These factors all indicate that this is the area in which automation could help the most, and as such will be the main focus of the case study going forwards.

3.2 Service Design and Factor Extraction

This section will present our approach to designing the risk assessment tool. It includes all design solutions that were considered and motivations as to why certain solutions were better suited than others. First we will explore the context and data available, we will then propose a way of extracting factors and labeling bugs. The results from these sections will then lead to a proposed way of implementing the service into a DevOps pipeline.

3.2.1 Context Evaluation and Data Exploration

It is inevitable that our assessment is going to have to be based on historical data. This makes data handling a central part of our proposed solution. There are three steps involved in our data exploration. First, we need to figure out what data sources are available. This will be based on the studies done in Section 3.1. Secondly we will outline what factors are likely to have an impact on risk, and that are machine extractable and readable. This will be based on the factors outlined in Section 2.2.4 and metrics proposed in 2.3. We will also allow ourselves some hypothesising. Thirdly, due to the limited time frame of the project, we are going to have to delimitate somewhat. This delimitation will be motivated, again, with our findings from sections 2.2.4 and 3.1.

Data Sources

The initial step was to identify where data can be extracted from, based on the tools used during the development pipeline discussed in Section 3.1.1. The GitLab API, local git metadata, and Jira were all investigated. The git metadata stored locally in the .git repository was chosen as the exclusive source of our data extraction. Early tests showed that performance was an issue when calling the API while traversing a large codebase, and the same data is available locally. JIRA is thought to be a way of increasing the accuracy of some parts of our service, but does not in of itself contain enough information to predict fairly. As such JIRA exploration will be left for future works and discussed in Section 5.2.3. One likely candidate for further extraction would be a static code quality metric analyzer. If this tool was able to provide machine-readable version of the factors outlined in section 3.2.2, it is hypothesised that this would impact risk assessment and be generalizable and algorithmically extractable. This, we believe, might be a good trail to go down in terms of future work, especially since it recently got decided that Sinch would add a static code reviewer to its stack, the program Sonarqube.

Likely Candidates for Extractable Risk Factors

As mentioned in section 3.1.2, interviews with developers and managers made it clear that it was implementation complexity in the RAS form that was by far the most difficult to estimate, while the other categories were often well known. Therefore, we will focus on factors that pertain to this area. Based on the factors outlined in Section 2.2.4, and our study of available data sources, this mock-up of a relational database was constructed. The fields are all factors available for automatic extraction, assuming the company implements a code quality metric tool, and thought to be relevant for the associated risk score of a commit. This diagram is shown in figure 3.2.

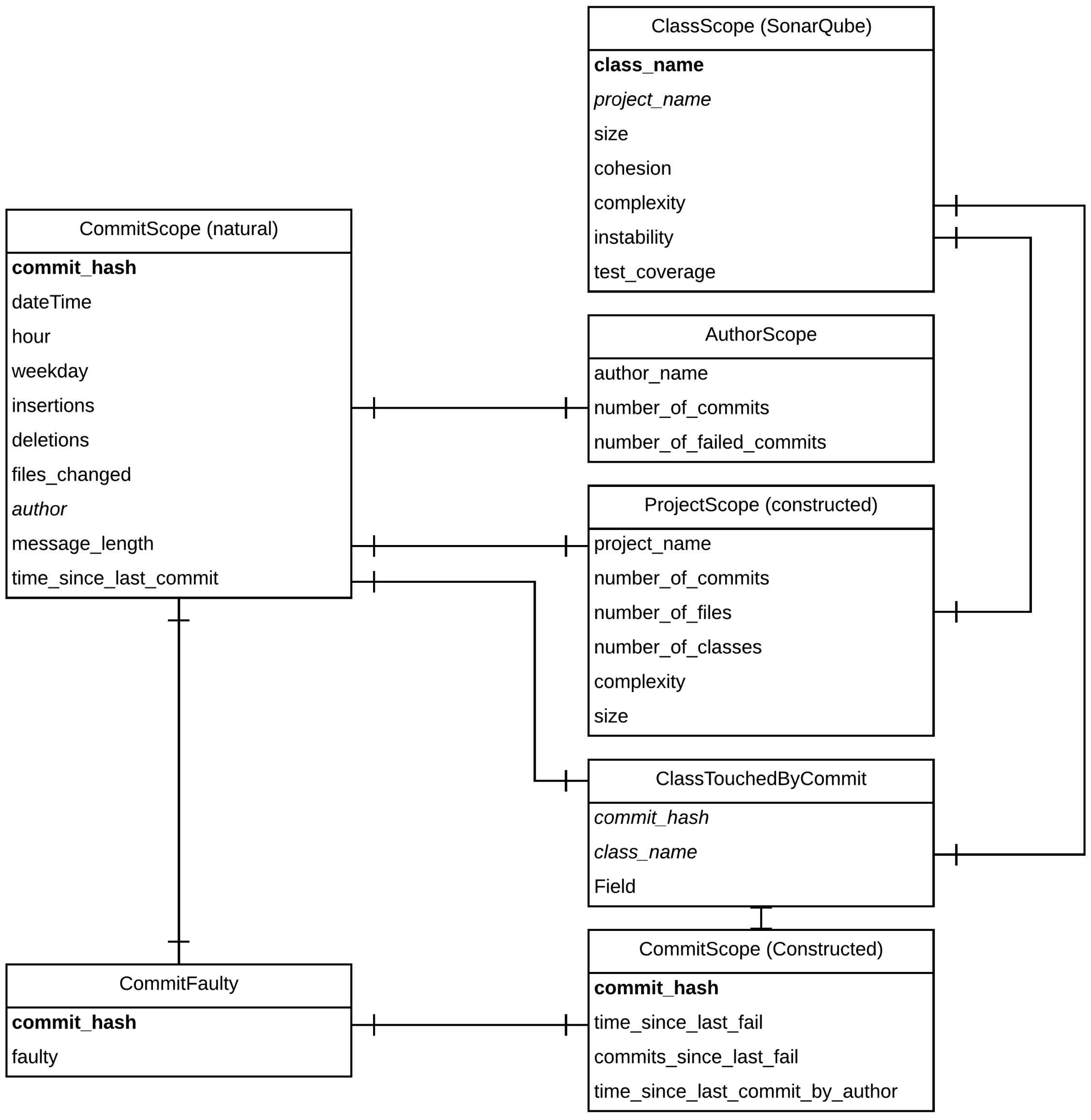


Figure 3.2: E/R diagram of the potentially extractable risk factors

Delimitation

In order to prove that the concept of a data-driven risk assessment tool works, it was decided to start with the most impactful, and easiest to extract data source. This is the data natively shown in git log and git show. In figure 3.2, this is denoted as CommitScope (natural). It is believed that a more extensive factor extraction, where all factors presented earlier in Section 2.2.4 are considered, and that used several data sources would make for a better risk assessment, although, but due to time constraints, this will be left for future work. The studying of risk related to individual authors, or teams is rather easy to implement, but carries some moral predicaments. Labeling one individual's contributions as riskier might have implications for morale, and the prospect of something resembling an internal competence ranking might affect what work developers are willing to take on. With these delimitations in mind, the final list of risk factors we are basing our predictions on will be listed in Section 3.2.2.

3.2.2 Factor Extraction

The ER diagram, presented in figure 3.2, illustrates the factors believed to have a potential impact on the risk of a commit, while also being systematically extractable and generalizable, given the circumstances specified in the case study. However, not all of these could be examined within the scope of our project, and as discussed in the previous section some further delimitation had to be imposed. Based on the reasoning explained there, factors on the right side of the diagram will be left as potential future additions. This mostly due to the lack of a designated code quality metric tool in Sinch's pipeline, and that the programmatical challenge of developing one ourselves would leave us with insufficient time for the other areas. Furthermore, some factors, like commits since last failed, increased the time complexity of the dataset population component of our tool significantly, making the end product too slow for its intended use case as a component in a CI/CD pipeline. Factors related to authors were also excluded, in part due to this being potentially sensitive information. However, with clever name obfuscation, this might be an area worth looking into in the future. Especially the number of commits a certain author has had in a project is thought to be of significance. Lastly, parsing weekdays into numerical values turned out to be problematic, and gave no conclusive results. This problem stems from the cyclic nature of this factor, where Sunday and Monday would be represented as furthest apart, even though they are chronologically very close to each other.

The factors ultimately chosen for extraction are therefore:

- **Time since last commit** (How long has it been since the last change was commited?)
- **Files Changed** (How many files have been changed?)
- **Insertions** (How many lines have been inserted?)
- **Deletions** (How many lines has been deleted?)
- **Commit Message Length** (How long is the commit message?)
- **Commit Hour** (What hour of the day did the change get commited?)

3.2.3 Labelling Bugs

In order to start exploring the connection between hypothesized risk factors and de facto adverse events, we had to devise a systematical approach of labeling these bugs, or faulty states, their impact, and what caused them to occur. As mentioned in Section 3.2, local git-repositories were chosen as the main basis for factor extraction, and subsequent risk assessment. Interviews have shown that the most important use case of the service is facilitating the grading of implementational complexity through statistically based prognoses. This grading is currently done on a deployment-level basis. An argument could therefore be made that we ought to study the deployments as entities, and try to fit a label on it as a whole. However, since git implements the change-set model, there are atomic elements, the commits. Because of this, deployments can be studied as a set of commits, rather than a standalone instance. This reduces the problem of assessing a whole deployment's risk, to assessing each individual commits risk score, and then aggregating them. This increases granularity and allows for much more numerous data points, something that allows for more extensive analysis. Stakeholders at Sinch also expressed an interest in displaying the individual commits' risk scores as a part of the desired functionality of the service. This information is believed to facilitate the reviewing part of the CAB-process, since commits with a high risk score demand more rigorous scrutiny, and vice versa. With all this in mind, it is highly likely that the labeling that will provide the greatest boon is on a commit level. Risk factors will be extracted from an individual commit through git metadata, and this commit will be labeled with an approximation of the adversity it caused.

In order to study how to best achieve this labeling, we were given access to a variety of Sinch projects in GitLab, making it possible to study their corresponding git-metatdata. A number of open-source projects from the GitLab ”explore” page were also considered. Two major approaches were outlined:

- **Approach 1**, manually traverse the codebase and gitlogs looking for faulty states, and then reverse trace their causes, and label these.
- **Approach 2**, design an algorithmic solution that approximates these connections.

The approaches each have their upsides and downsides. Approach one can be much more granular and easily adaptable to the structure of any project. There are almost no limitations on what can be labeled. However, it quickly became clear that in order to be in any way generalizable, this labeling process was too time-consuming, and required project-wide, domain-level knowledge. It became clear that an algorithmic approach had to be devised. Based of how a human would approach this we made a list of activities that had to be performed in order to approximate whether a commit induced a faulty state or not.

A faulty state was defined according to the explanation in Section 2.2.1: “a given bug was introduced by the lines of code that were modifed to fix it”. The process of labeling faulty commits then followed the four-step method presented below, which was heavily inspired by Hinnefeld's work [26].

Step one: Gather all the git commits whose commit messages include the words ”bug” and ”revert”. These words are based on Hinnefeld's study, and have been tested to best capture bug fixes. ”Fixed”, however, resulted in too many false positives in our use case and was therefore excluded.

```
git log -i -all -grep BUG -grep FIXED -grep REVERT -pretty=format:%h
```

Step two: Examine which file(s) have been affected by a specific commit. Do this for every fault fixing commit.

```
git -no-pager diff [commitsha] [commitsha]^-name-only
```

Step three: Extract the line numbers of all lines that were affected by a certain commit in a certain file.

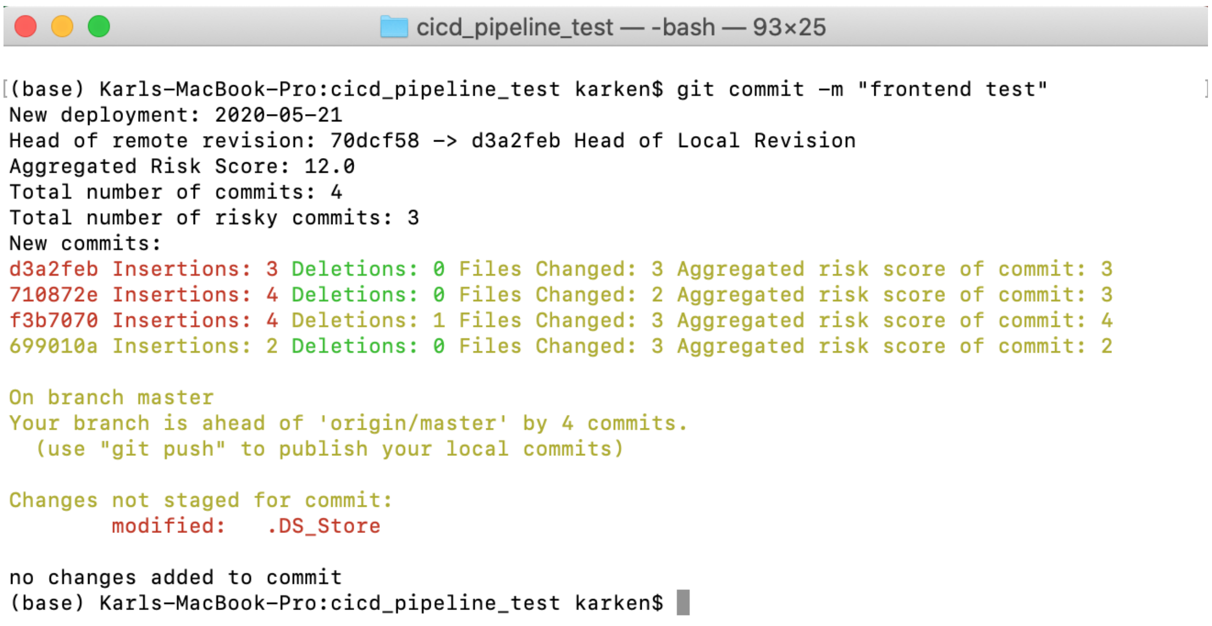
```
git -no-pager diff commitĉommit -U0 - fname
```

Step four: Get the commit that last touched the lines extracted in the previous step. This is the commit that was responsible for the lines that were fixed in the bug fix, and is therefore assumed to be the bug inducing commit.

```
git -no-pager blame -Lstart,+n commit^- fname
```

3.2.4 DevOps integration

In order for our research to substantiate actual value created, it is important that the service is well integrated into the company's workflow. This means that it happens at the right time in the development process, and confers enough clarity to be a basis for data-driven decisions. It is also important that the tool does not overburden the stakeholders with unnecessary information, and that the key metrics are emphasized. In accordance to figure 2.1 (The DevOps eight), we will build the model based on data fetched in the monitoring step. For our case, this is the metadata stored in git. The model will then be called just before the deployment step, in order to give an assessment. This is easiest solved with a two-pronged program, that has one part that trains the model, and one part that is called just before a new deployment. This all should be packaged as a GitLab component that can be integrated into the GitLab CI/CD pipeline. The assessment part of the program shall be called by a git hook, a job-based approach that calls the program as a step of the deployment process. A prototype was created to be able to visualize how the model's predictions could be displayed, based on the findings in Section 2.2.5. After iterations with stakeholders, the illustration shown in Figure 3.3 was decided as a good starting point.



```
(base) Karls-MacBook-Pro:cicd_pipeline_test karken$ git commit -m "frontend test"
New deployment: 2020-05-21
Head of remote revision: 70dcf58 -> d3a2feb Head of Local Revision
Aggregated Risk Score: 12.0
Total number of commits: 4
Total number of risky commits: 3
New commits:
d3a2feb Insertions: 3 Deletions: 0 Files Changed: 3 Aggregated risk score of commit: 3
710872e Insertions: 4 Deletions: 0 Files Changed: 2 Aggregated risk score of commit: 3
f3b7070 Insertions: 4 Deletions: 1 Files Changed: 3 Aggregated risk score of commit: 4
699010a Insertions: 2 Deletions: 0 Files Changed: 3 Aggregated risk score of commit: 2

On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
        modified:   .DS_Store

no changes added to commit
(base) Karls-MacBook-Pro:cicd_pipeline_test karken$
```

Figure 3.3: Frontend Prototype

Here, some cardinal values of factors are displayed. Red is a high-risk value, yellow is medium, green is low. These are based on the recommendations mentioned in section 2.2.5, under Risk Representation. The git hash of the commit in question also gets the color of the factor with the highest risk. In reviewing, starting with red commits might be a way of facilitating the work. Every commit also has a related risk score. This is the sum of all its commits. the weighing of this sum will be studied further after results are analyzed. There will also be an aggregated score for the whole deployment, as of right now it is just a sum of the commit's scores. Interviews show that this already is an improvement over the current process, but it is hoped that an even better one will be found during the analysis in Section 5.1.

3.3 Applied Factor Risk Impact Analysis

In this section, the labeled data and extracted factors are utilized to study a potential correlation. Two main approaches were identified. Our first attempt was studying plots and applying statistically based methods such as checking averages, t-distribution analysis, and k-means-clustering. It was believed that static analysis would allow for factor weighing that could lead to a fair prediction. The second approach was applying various machine learning methods, mostly shallow learning models, in order to predict the correlations.

3.3.1 Manual Statistical Analysis

Firstly, the output from the factor extraction and bug labeling was divided into two data matrices, one containing the faulty commits and one containing the non-faulty commits. Each row in the respective matrix represented a commit and the columns represented the risk factor values.

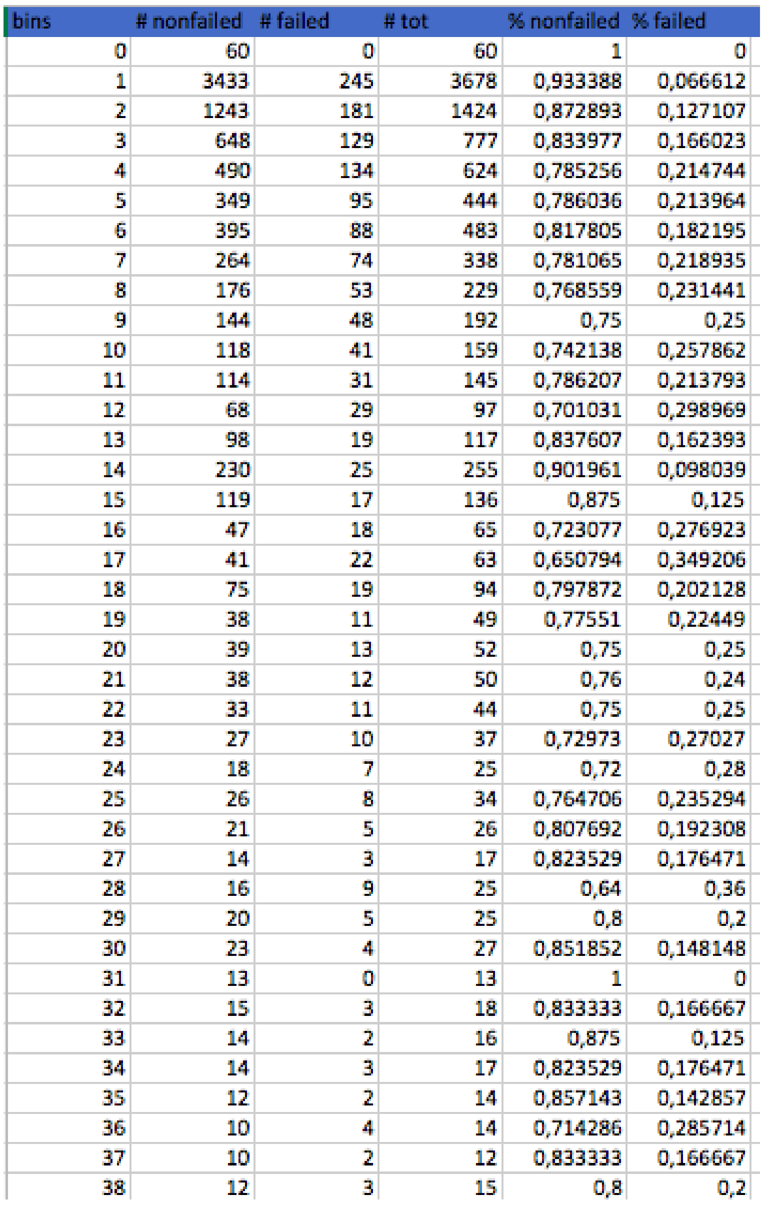
This separation was done in python, which enabled simple data handling. Subsequently, the two data matrices were analyzed in Excel. Each factor value was plotted for all commits, and a visual comparison between the failed and non-failed commits was constructed. This was meant to provide a simple overview of whether or not there were any differences. By simply observing the scatter graphs, no distinctive correlation could be seen. Results are presented in section 4.1 and Appendix B. In order to compare the faulty and non-faulty commit factor values numerically, the average values were obtained for each column with the excel function "average". All factor averages were higher for the faulty commits, compared to the non-faulty. However, in order to determine if any of the differences were statistically significant, the factors' two-tailed student t-distributions of their respective means were plotted. The upper and lower bounds were calculated according to equation 3.1 and 3.2:

$$U = \overline{x} + t * s/\sqrt{n} \tag{3.1}$$

$$L = \overline{x} - t * s/\sqrt{n} \tag{3.2}$$

Where U is the upper bound, L is the lower bound, s denotes the sampled standard deviation and t is the two-tailed t-distribution value for a 5% significance level and n-1 degrees of freedom. S was obtained by the Excel function "stdev" and t from "T.INV.2T". The null hypothesis was that the non-faulty and faulty commit factors had the same mean. This hypothesis could be rejected with 95% confidence if the boundaries did not overlap. If this were the case, it would be a big indication that these factors have an influence on the likelihood of failure of a commit.

The aim was to mimic the scoring system of 0, 1, 2 and 3 used by Sinch to classify the risk. This was because only one risk dimension of the RAS-form was being analyzed, namely the implementation complexity, and it would be beneficial to have all risk scores on the same format. The idea was therefore to identify risk levels, from 0-3, for each individual factor and then calculating the accumulated risk score by applying weights, as performed in the SRAM model in Section 2.2.3. An attempt to cluster the data into 4 categories for each factor was therefore done using the k-means method with 4 centers. The k-means clustering was performed in r using the method "kmeans" with 4 centers, 50 maximum iterations, and nstart = 50. This resulted in 4 groups of data, which represented the different risk levels. However, when further analyzing the groups identified by the k-means method, it was clear that the results were inconclusive. Further analysis was done by calculating the percentage of failed versus non-failed commits for every value of a factor. For example, looking at "number of files changed", the values were (0, 1, 2, 3 ... max number of files changed). The frequency of failed and non-failed commits for every value was then obtained by the FREQUENCY function in Excel. Their respective percentages could then easily be calculated by dividing the frequency with the total number of commits that changed the same number of files. An extract from the excel spreadsheet is presented below in Figure 3.4 to further demonstrate. The "bins" column is the number of files changed by a commit and the values in " nonfailed", " failed" and " tot" are the number of commits that did not fail, the number that failed and the total number of commits that changed a certain number of files.



bins	# nonfailed	# failed	# tot	% nonfailed	% failed
0	60	0	60	1	0
1	3433	245	3678	0,933388	0,066612
2	1243	181	1424	0,872893	0,127107
3	648	129	777	0,833977	0,166023
4	490	134	624	0,785256	0,214744
5	349	95	444	0,786036	0,213964
6	395	88	483	0,817805	0,182195
7	264	74	338	0,781065	0,218935
8	176	53	229	0,768559	0,231441
9	144	48	192	0,75	0,25
10	118	41	159	0,742138	0,257862
11	114	31	145	0,786207	0,213793
12	68	29	97	0,701031	0,298969
13	98	19	117	0,837607	0,162393
14	230	25	255	0,901961	0,098039
15	119	17	136	0,875	0,125
16	47	18	65	0,723077	0,276923
17	41	22	63	0,650794	0,349206
18	75	19	94	0,797872	0,202128
19	38	11	49	0,77551	0,22449
20	39	13	52	0,75	0,25
21	38	12	50	0,76	0,24
22	33	11	44	0,75	0,25
23	27	10	37	0,72973	0,27027
24	18	7	25	0,72	0,28
25	26	8	34	0,764706	0,235294
26	21	5	26	0,807692	0,192308
27	14	3	17	0,823529	0,176471
28	16	9	25	0,64	0,36
29	20	5	25	0,8	0,2
30	23	4	27	0,851852	0,148148
31	13	0	13	1	0
32	15	3	18	0,833333	0,166667
33	14	2	16	0,875	0,125
34	14	3	17	0,823529	0,176471
35	12	2	14	0,857143	0,142857
36	10	4	14	0,714286	0,285714
37	10	2	12	0,833333	0,166667
38	12	3	15	0,8	0,2

Figure 3.4: Percentage failed vs non-failed for every value of files changed

As can be seen, the percentage of failed commits are initially quickly trending upwards when increasing the number of files changed, but after around 13 files changed, it becomes difficult to draw any solid conclusions. As mentioned in Section 2.2, the IRM recommends the following risk levels: low (2% probability of failure), medium (2-25% probability of failure) and high (25+% probability of failure). According to this recommendation, the levels obtained from the k-means clustering were not valid. The resulting k-means cluster means and their respective group sizes can be seen in Table 3.1.

Cluster Mean	Number of Commits in Cluster
542.8	4
250.4	8
80.34	58
7.376	1397

Table 3.1: Showing the resulting cluster means from the k-means clustering, and their respective group sizes

Because of the inconclusive nature of the clustering. An attempt to manually assign the levels 0-3 was done by looking at the percentage of failure and deciding cut-off points, influenced by the IRM recommendation mentioned in Section 2.2. However, this proved to be difficult and it also lacked sufficient scientific backing. The task of determining the input-output relations was instead deemed to be a use case suited for machine learning, as multiple benefits could be seen by applying it. Firstly, potential synergy effects between the various factors would be considered, as opposed to when dividing the individual factors into risk levels. The precision of the prediction would also increase since the factor value would not be part of a range that corresponds to the same risk score. Lastly, the issue of determining weights for the different factors would no longer be a problem, as this would be produced by the machine learning algorithm.

3.4 Machine Learning

The end result of the factor extraction is a table of commits hashes and their respective cardinal risk factor scores. The result of the bug labeling is an approximation on whether the commit caused a bug or not. This data is already in a format that is close to optimal when training a binary classifier. There are, however some considerations and limitations. One observation about the data is that it is rather unbalanced. This means that one of the classes, in our case the non-bug-inducing class, is vastly over-represented, in the case of MSL with an approximately 7:1 ratio. This can lead to optimistic results if only looking at the general precision (total number of right predictions over the total number of predictions). This effect stems from that for a data-set like this, the best model might be one that guesses "not a bug" no matter the feature values. In our case, this would lead to an accuracy of about 85%, which might sound impressive, but that in effect could be replaced with a flag set to "not a bug" for all the commits. It is also probable that we want the model to be stringent in the sense that additional false positives matter less than bug inducing commits that slip through unnoticed. This can be changed by tuning the threshold parameter of the model. Another consideration is that given the non-decisive nature of our data-set, as even the smallest commit can potentially cause a bug, it is likely that the main takeaway from our model should not be a fully classified data-set without additional information, but rather a fuzzy value prediction of how likely the commit is to induce a bug. In aggregated form, it is likely that this would give a better classification of the risk associated with a full deployment. Visualization of this data would also give a clear prioritisation of how to review the commits, where high scoring commits could be reviewed first, or more rigorously. In order to gauge how good a model is at this, a commonly used metric is the ROC-curve, and its associated AUC-value, described in section 2.4, under Performance Metrics. Another consideration to factor in is the model training time. As discussed in chapter 2.1, DevOps, the component needs to be lightweight in order to add value. As such, model training time in the order of seconds, rather than minutes, even without access to a company-issued GPU, is beneficial.

In order to evaluate the feasibility of a machine learning-based approach to the issue of risk factor impact analysis, four different models were trained on data extracted from the MSL project. This is the biggest project at Sinch, and also the one we are the most familiar with. For a use case this datalogically simple (one cardinal feature vector matched to one one-hot encoded vector), shallow learning models are sufficient. This also allows for a deeper understanding of how the model works, something that allows for further analysis. Four models were selected as likely candidates.

- LinearSVC
- Logistic Regression
- Random Forest Classifier
- Naive Bayes

All of them were downloaded from the SKLearn python libraries. More info on the details of how the models function can be found in the SKlearn documentation [48]. All these models were trained on 90% of the data extracted from the MSL project. They were then tested against the other 10%. This split was based on chronological order, as to simulate the company having started using the product 90% of the way into the project, and then testing the results on the most recent 10%. This split is crudely illustrated in figure 3.5.

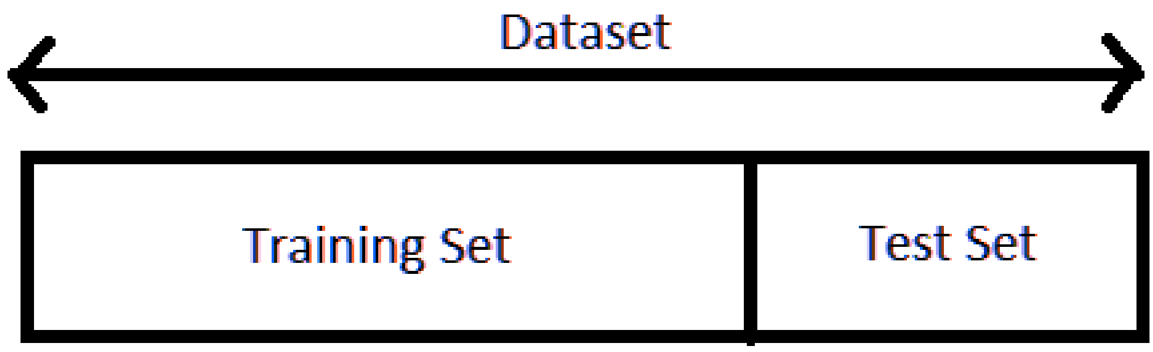


Figure 3.5: A dataset split into training and test data [40]

In order to study the performance of the models, some of the metrics mentioned in Section 2.4.2 will be of extra interest. The models's respective confusion matrices will be displayed, both with the default threshold and with a threshold that allows the model to catch approximately 66% of the bugs. Furthermore, the ROC-curve, and its associated AUC-score will be graphed. This is believed to be the fairest representation of how well the model predicts and as such the most important performance indicator. We also compared the relative speeds of the training process. Results shown in Section 4, makes it likely that at least random forest classification had a sufficient understanding of the correlations between risk factors and adverse events to be of assistance in the assessment process. Tuning hyperparameters such as increasing the number of estimators made the result slightly more reliable, at the cost of training being slightly more computationally expensive. Even for a number of estimators that trains as quickly as the other models, random forest classification presented the best results. In order to study the generalizability of our results, the factor extraction, bug labeling, and machine learning model were tested on other projects, both open-source, and Sinch projects. The importance of the extracted factors was also studied through the trained models' built-in feature weighing functionality.

Chapter 4
Results

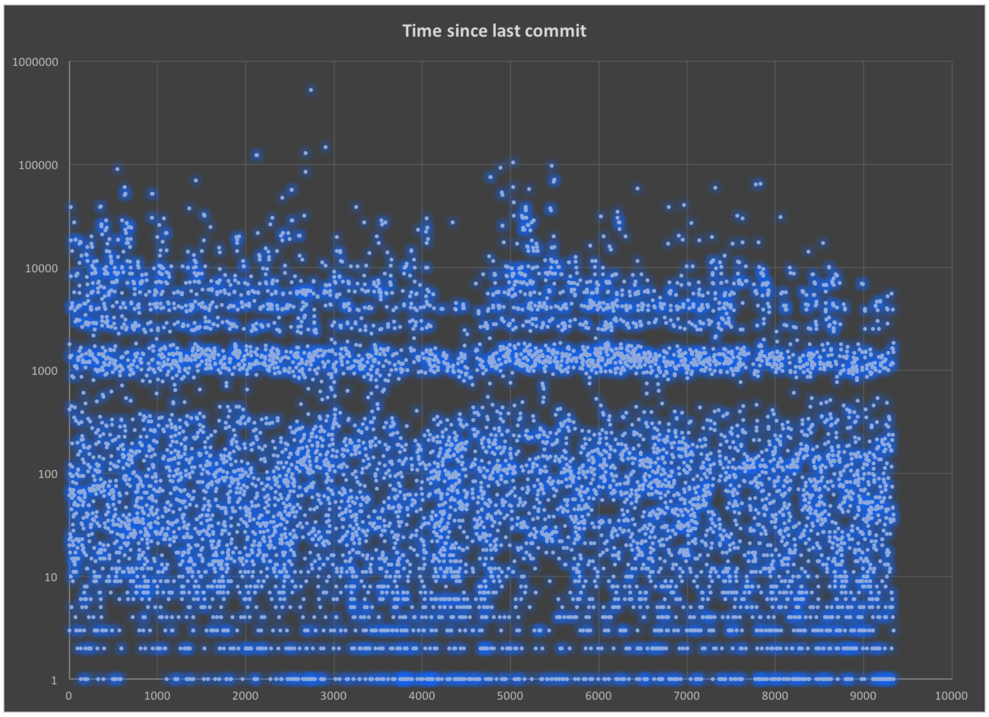
This section will present and summarize the results of the tests conducted. All of the tests are based on the datasets created through the factor extraction and bug labeling steps discussed in section 3.2.2 and 3.2.3. There will be two main areas from which results are of importance. The first being our statistical analysis on the MSL dataset. The second is the results from the application of shallow learning methods and machine learning, both several model's performances on the MSL dataset, and later, Random Forest Classifiers performance on other projects.

4.1 Manual Statistical Analysis - Results

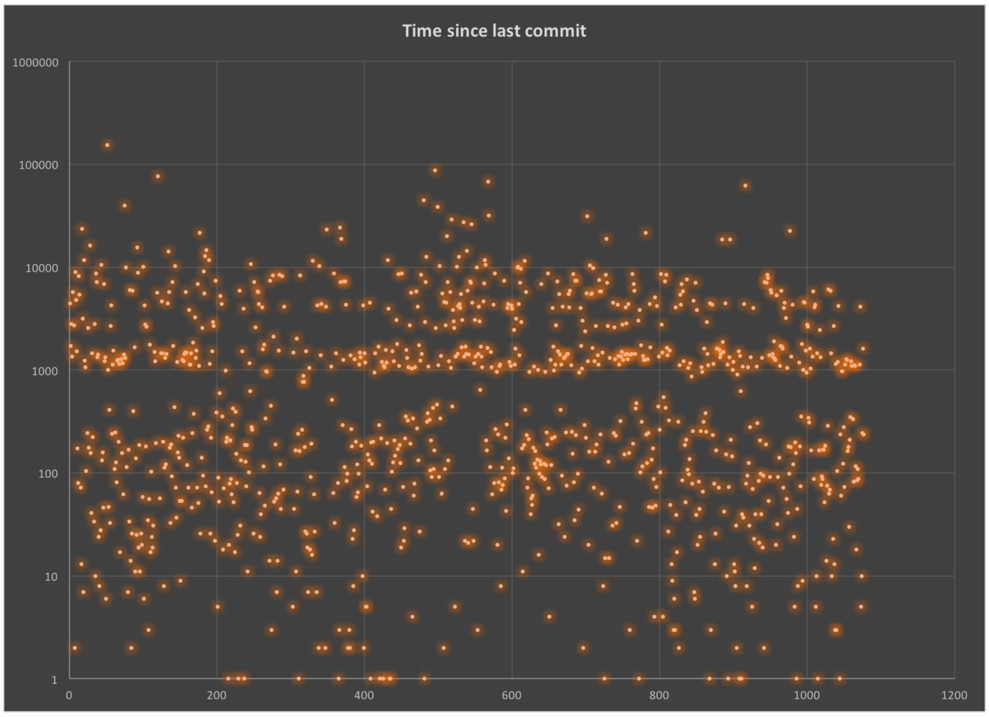
Here the results from the manual analysis will be displayed. These are based on two datasets. One with only faulty commits, and one with non-faulty commits. First, scatter plots of the two datasets' factor value ”time since last commit” for every commit will be displayed. After that, the averages and medians for the factor values in the two datasets will be contrasted. Lastly, we will study the statistical difference by calculating the true distribution using a student t-test.

Scatter Plots

The results of the scatter plots for the various risk factors where rather inconclusive and comprehensive. Hence, we have chosen to only display the results from the factor ”time since last commit”, in Figure 4.1. It has the the most distinguishable difference. The rest of the factors' scatter plots can be found in Appendix B.



(a) Non-faulty commits



(b) Faulty commits

Figure 4.1: All individual dots in (a) and (b) illustrate a commit. The y-axis represent the time since last commit in seconds and the x-axis are simply the different commits

Risk Factor Averages and Medians

The average values and the median of the risk factors for faulty and non-faulty commits are compiled in Table 4.1 and 4.2 below.

Average Factor Values			
Factor	**Non-faulty Commits**	**Faulty Commits**	**%Difference**
Commit hour	12,67	12,97	2,185
Files changed	9,620	13,05	35,61
Time since last commit	1723	2288	32,83
Message length	74,78	87,75	17,34
Insertions	279,2	383,8	37,48
Deletions	176,0	262,1	48,95

Table 4.1: Illustrates the average factor values for non-faulty and faulty commits, as well as the percentual difference between them.

Median of Factor Values			
Factor	**Non-faulty Commits**	**Faulty Commits**	**%Difference**
Commit hour	13	13	0,000
Files changed	2	5	150,0
Time since last commit	68	251	269,1
Message length	56	49	-12,50
Insertions	15	72	380,0
Deletions	7	25	257,1

Table 4.2: Illustrates the median factor values for non-faulty and faulty commits, as well as the percentual difference between them.

Student-t Test of Factor Means

As can be seen, all averages for the faulty commits are higher than for the non-faulty commits. The median is also higher for all factors except for commit hour and message length.

In order to investigate if the differences seen in Table 4.1 were statistically significant, a student t-test was performed. In order to clearly illustrate the ranges in which the true mean lies, with 95% confidence, the upper and lower bounds of the means' t-distribution were calculated, as described in Section 3.3.1. If the factor values' ranges do not overlap, it is the equivalent of a p-test showing a value of 0.05 or lower, meaning that performing a p-test, in addition to the boundary calculation, would be superfluous. Table 4.3 and 4.4 shows the results from this calculation.

Non-faulty Commits' Student-t Test						
	Commit Hour	Files Changed	Time since last commit	Message length	Insertions	Deletions
Std Deviation	2,895	33,36	8151	95,65	2027	1946
n	8955	8955	8955	8955	8955	8955
Significance	0,05	0,05	0,05	0,05	0,05	0,05
Critical t	1,960	1,960	1,960	1,960	1,960	1,960
Lower limit	12,63	8,93	1554	72,80	237,2	135,7
Upper limit	12,75	10,31	1892	76,76	321,2	216,3

Table 4.3: Presents the variables needed to calculate the upper and lower limits of the non-faulty commits' factor means, shown in the highlighted cells

Faulty Commits' Student-t Test						
	Commit Hour	Files Changed	Time since last commit	Message length	Insertions	Deletions
Std Deviation	2,796	37,96	6850	137,9	2496	2115
n	1467	1467	1467	1467	1467	1467
Significance	0,05	0,05	0,05	0,05	0,05	0,05
Critical t	1,960	1,960	1,960	1,960	1,960	1,960
Lower limit	12,82	11,10	1938	80,69	256,0	153,8
Upper limit	13,11	14,99	2639	94,82	511,6	370,4

Table 4.4: Presents the variables needed to calculate the upper and lower limits of the faulty commits' factor means, shown in the highlighted cells.

Looking at the upper and lower limits for the different factor means, we see that commit hour, files changed, time since last commit, and message length don't have overlapping boundaries. The result of this is that we can reject the null hypothesis, that these factors have the same mean, with 95% confidence. Also, the factor means are consistently higher for the faulty commits compared to the non-faulty, implying that faulty commits generally have a later commit hour, more files changed, longer time since last commit, and a longer message length. For insertions and deletions, we cannot conclude anything with this level of significance. Only with a significance level of 23% we get non-overlapping boundaries for insertions, and for deletions we need a significance level of 26%. To clarify, this result is compiled in Table 4.5 below.

Factor	non-faulty commit boundaries (of true mean)	faulty commit boundaries (of true mean)	statistically significant at 0.05 significance level
Commit Hour	12.63 - 12.75	12.82 - 13.11	YES
Files Changed	8.93 - 10.31	11.10 - 14.99	YES
Time Since Last Commit	1554 - 1892	1938 - 2639	YES
Message Length	72.80 - 76.76	80.69 - 94.82	YES
Insertions	237.2 - 321.2	256.0 - 511.6	NO
Deletions	135.7 - 216.3	153.8 - 370.4	NO

Table 4.5: A compiled table showing what factors were statistically significant at 0.05 significance level

4.2 Model Performance on MSL

As mentioned in Section 3.4, four models were all trained on the data extracted from the MSL project and then compared. The metrics tested for were a confusion matrix, both with the default cutoff threshold, and one that catches 60% of the bugs, and a ROC-curve with the AUC-score written into it. Time of convergence was negligible in all the models, even on the larger projects. The four models tested are:

- **Support Vector Classifier**
- **Logistical Regression**
- **Naive Bayes**
- **Random Forest Classifier**

4.2.1 Support Vector Classifier

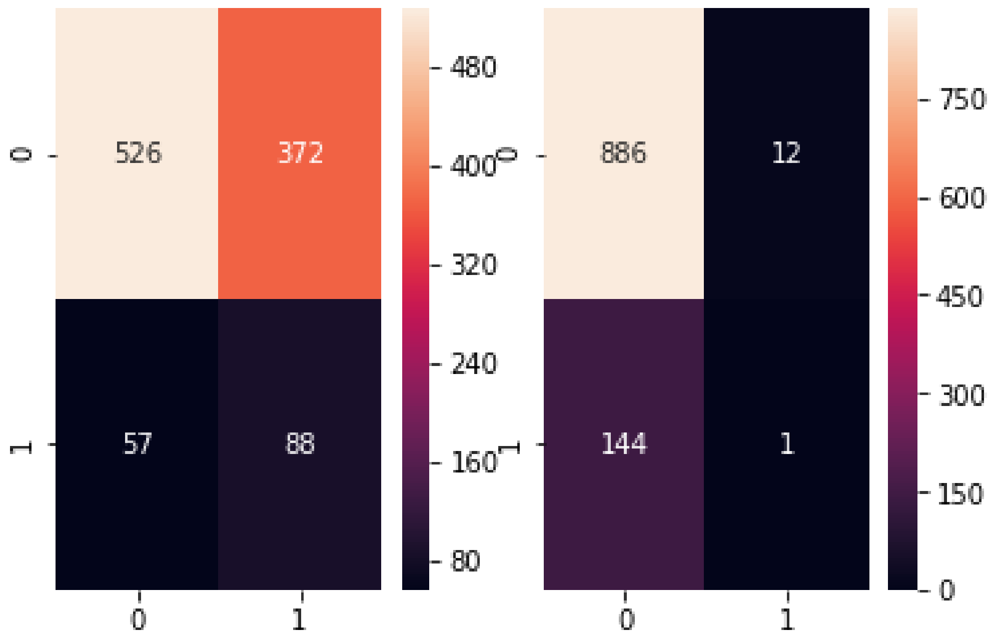


Figure 4.2: Confusion matrices for the support vector classifier. The naive case guesses that almost all events are non-buggy. The tuned case has decent performance, but a high number of false positives

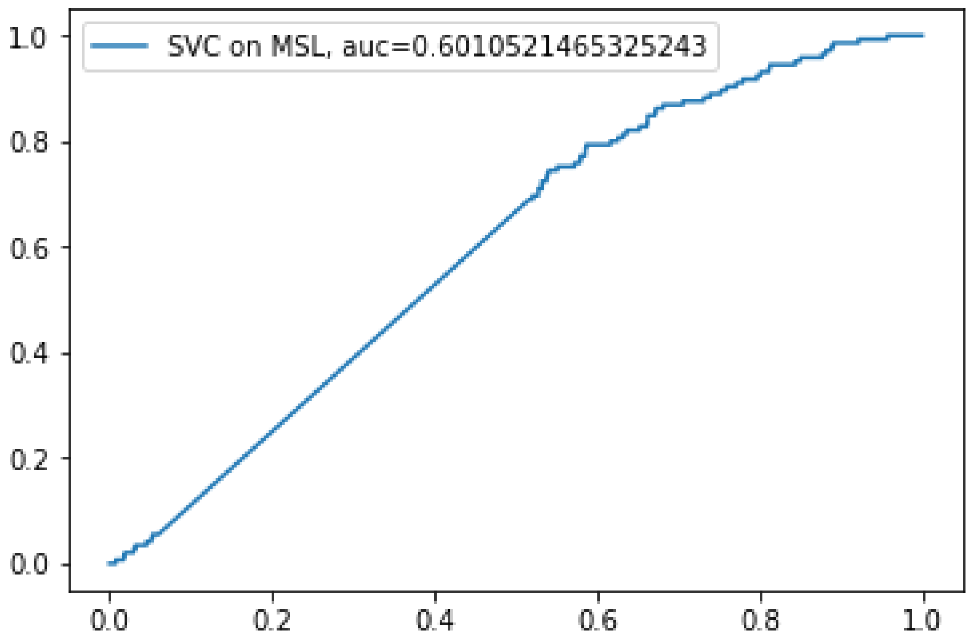


Figure 4.3: The ROC-curve for SVC on MSL. Relatively linear. Close to the baseline performance

Logistical Regression

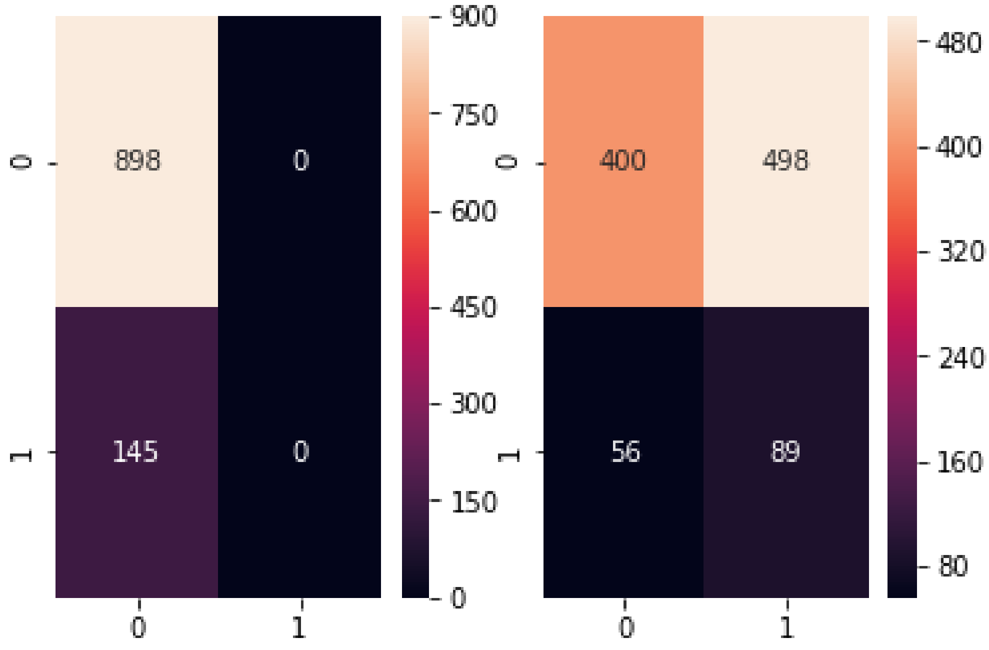


Figure 4.4: Confusion matrices for Logistical Regression on MSL. The naive case guesses that all events are non-buggy. The tuned one has more false negatives than true negatives.

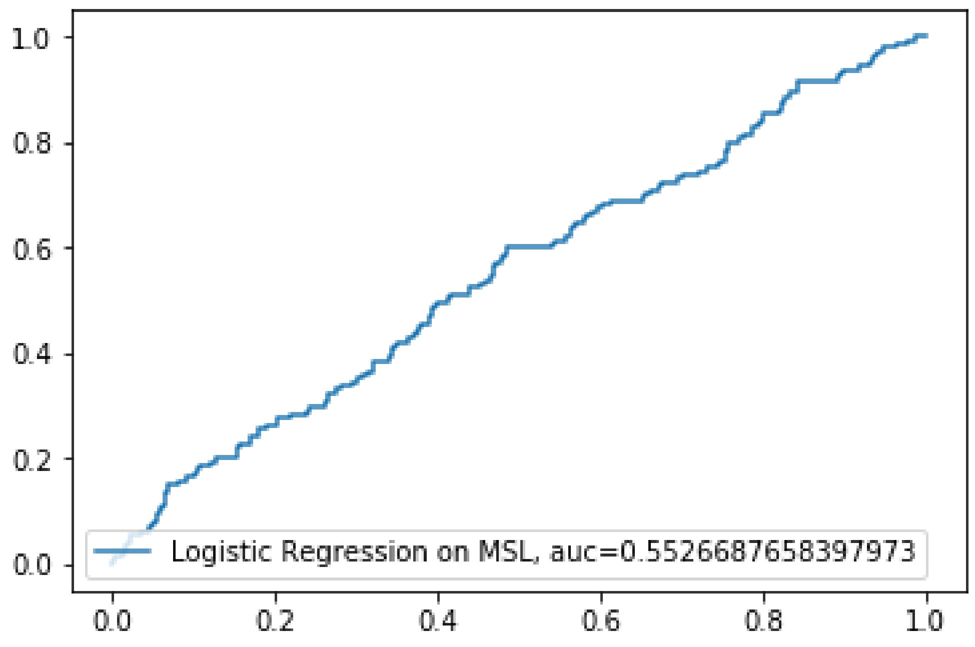


Figure 4.5: The ROC-curve for LR on MSL. Very close to the baseline model.

4.2.2 Naive Bayes

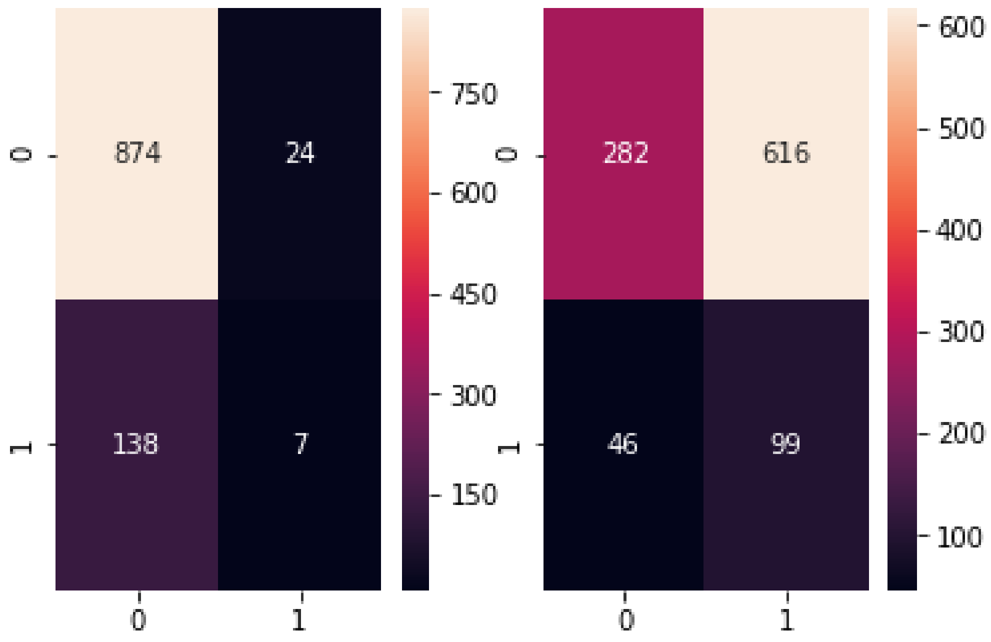


Figure 4.6: Heatmaps for NB on MSL, both performing rather poorly

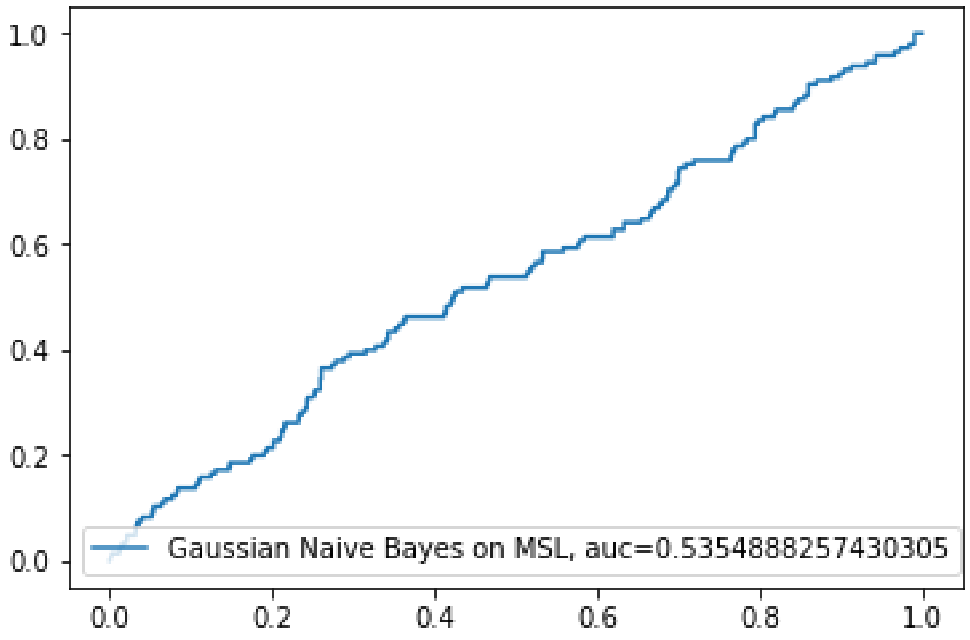


Figure 4.7: ROC-curve for NB on MSL

4.2.3 Random Forest Classifier

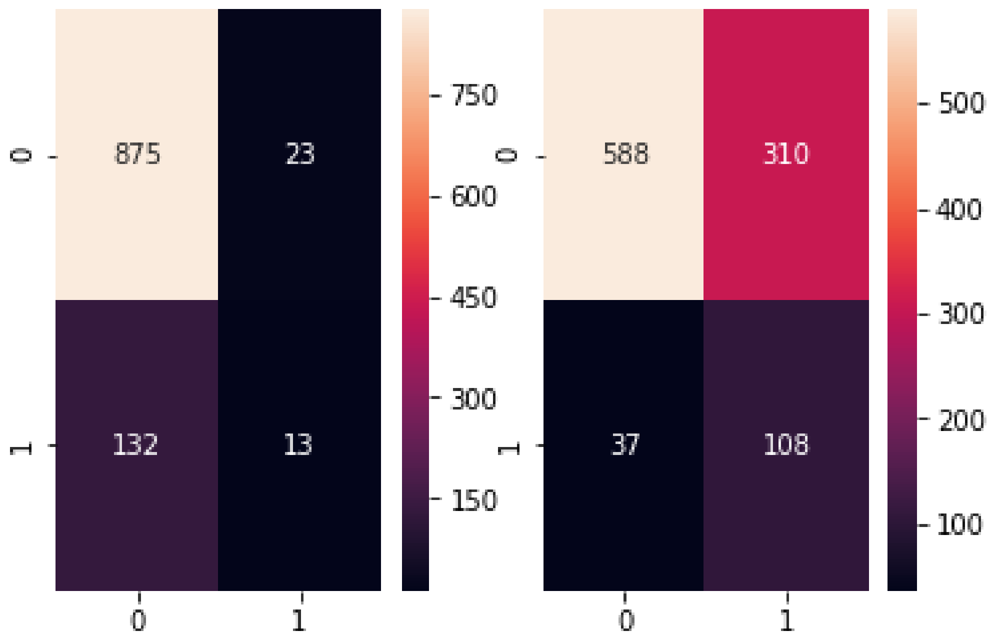


Figure 4.8: Confusion matrices for Random Forest Classifier on MSL. The naive one spots more true positives than previous models, but is still heavily biased towards negative results. The tuned one performs decently

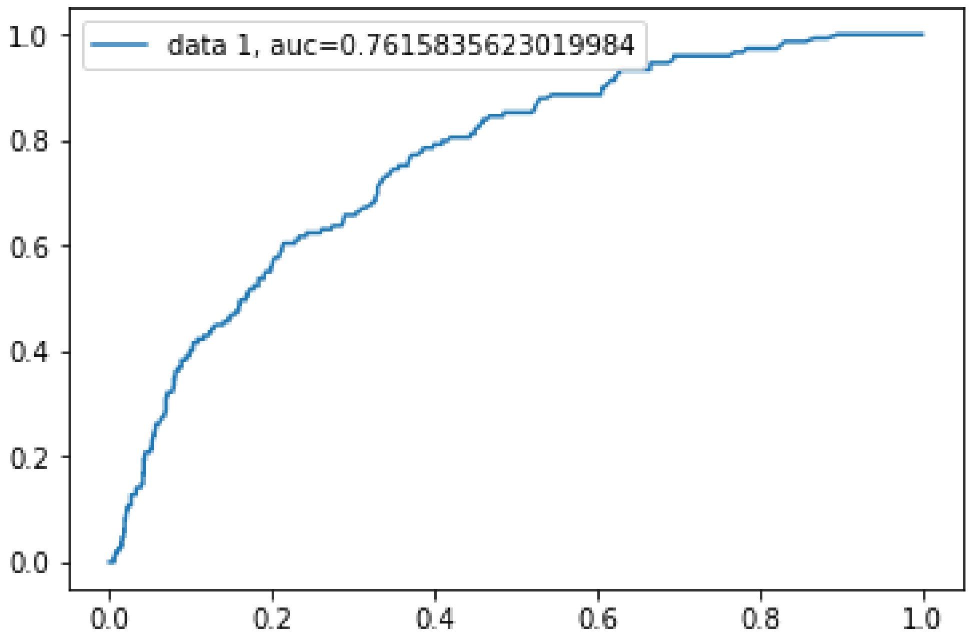


Figure 4.9: The ROC-curve for RFC on MSL. A clear improvement from the baseline model.

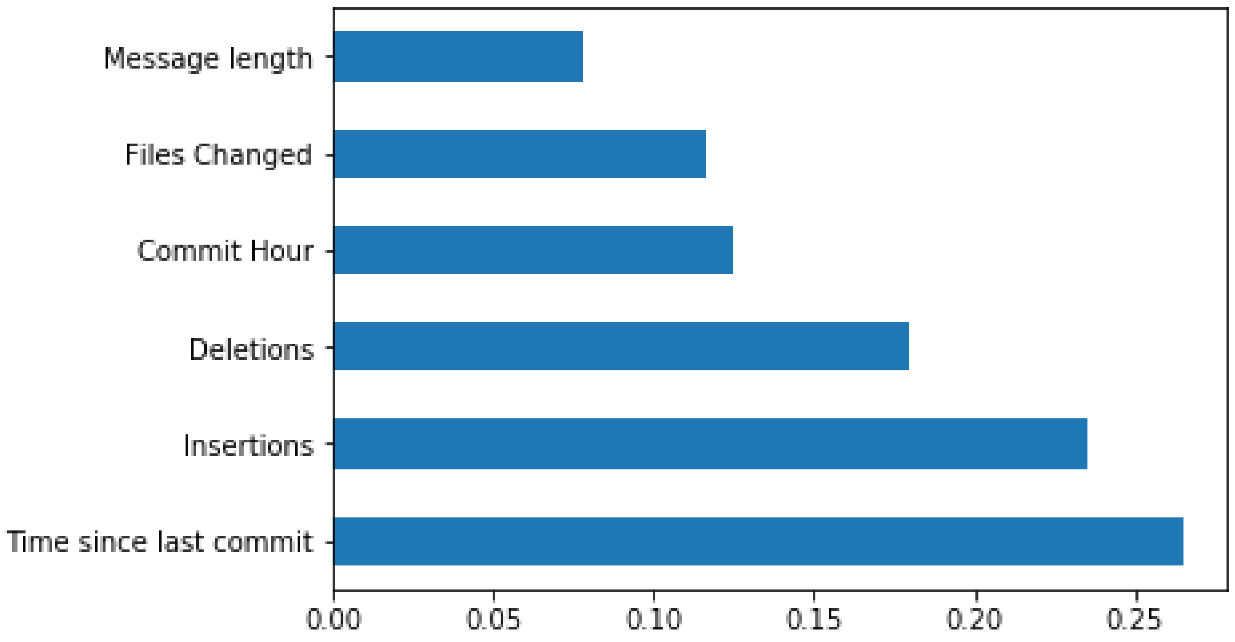


Figure 4.10: Feature Importance ranked for the RFC on MSL. This ranks the statistical impact of the factors on the model's prediction

4.3 Performance on Other Projects

Random forest classifier was elected as the most likely candidate for best model performance. In order to study how generalizable our findings are, we we trained the model on other open-source, git-based projects, and studied the same performance metrics as when electing what model to use.

4.3.1 Veloren

Veloren is a medium-sized open source project (4000 commits). It is a game with developers all over the world.

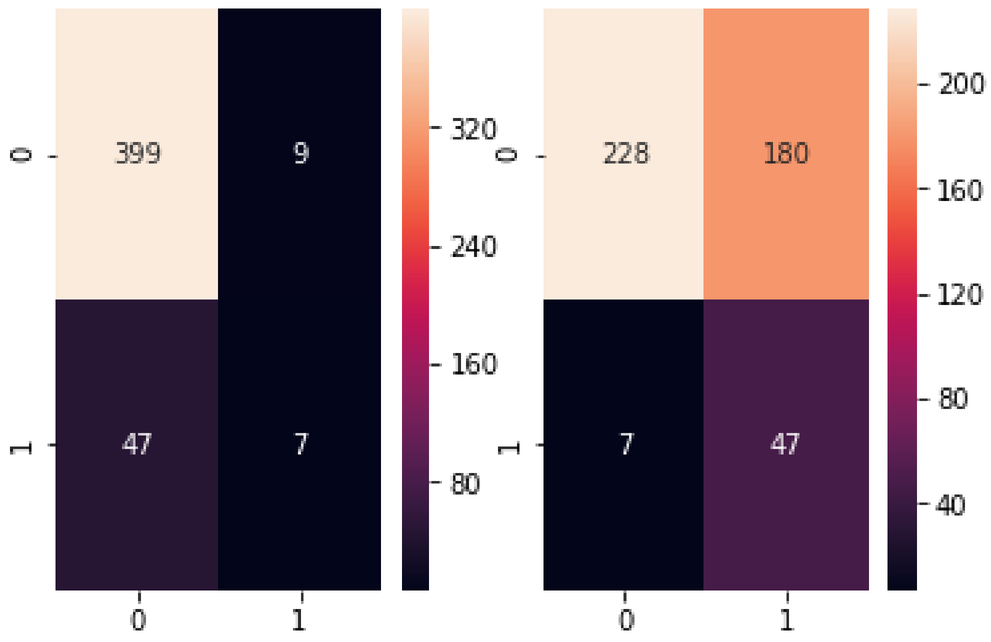


Figure 4.11: Confusion matrices for a model trained on Veloren. Naive one is heavily biased towards negative prediction. Tuned one performs decently

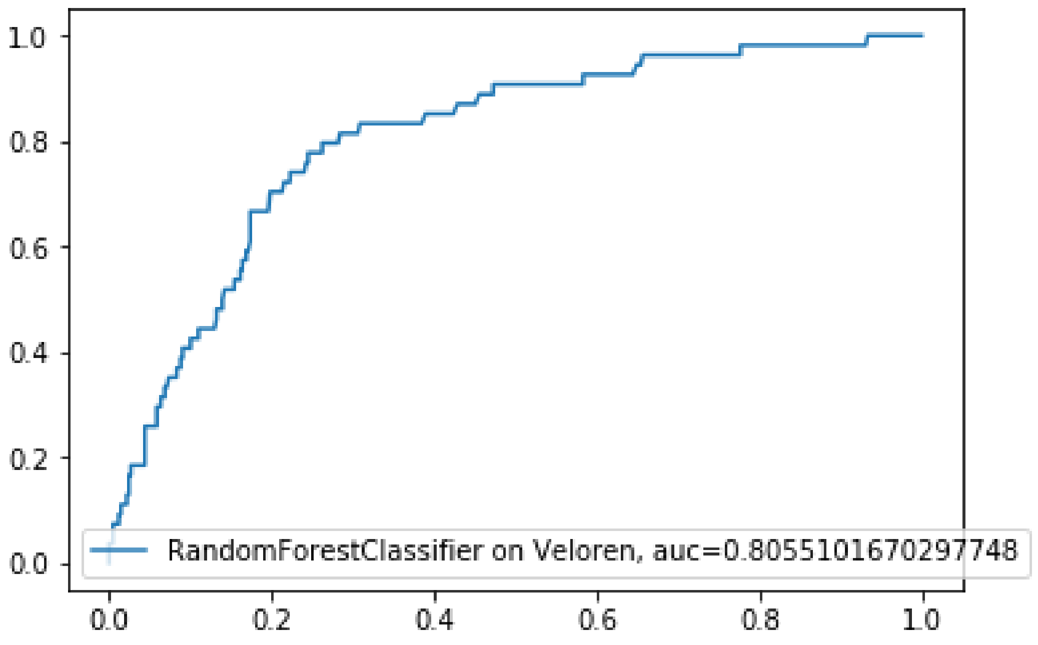


Figure 4.12: The ROC-curve for RFC trained on Veloren. Clear improvement from baseline

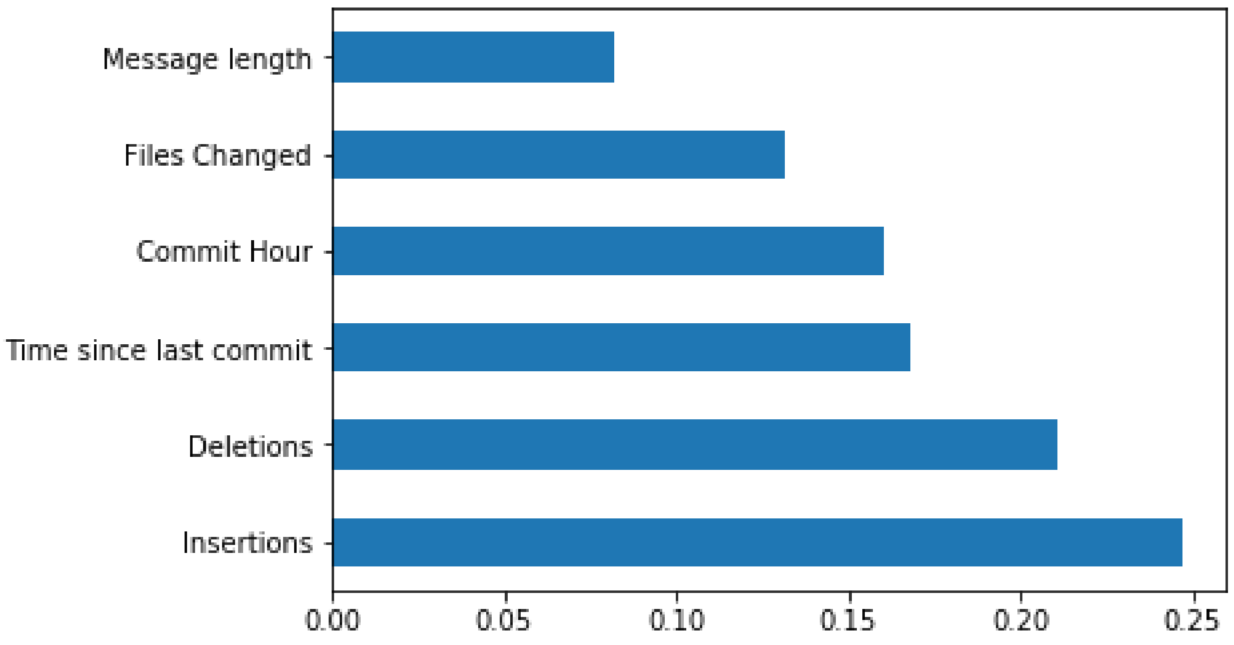


Figure 4.13: Feature importance, ranked from lowest impact to highest.

4.3.2 GitLab Omnibus

GitLab Omnibus is a large-scale (11500 commits) open-source project. It creates downloadable full-stack packages for various platforms. It has a relatively low ratio of bugs to commits compared to the other projects studied.

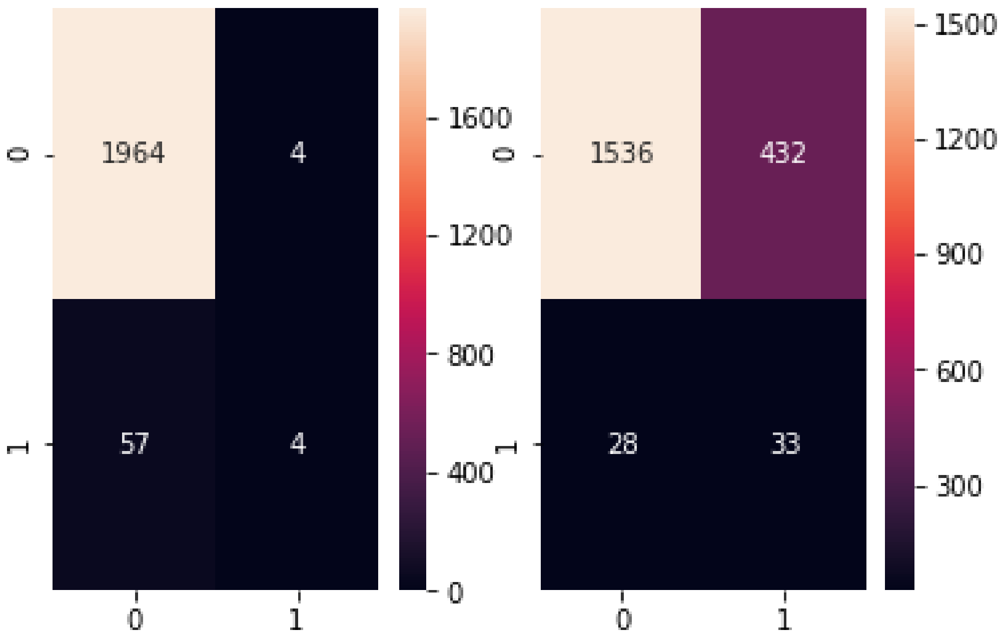


Figure 4.14: Confusion matrices for RFC on gitlab-omnibus. Even naive one spots some true positives.

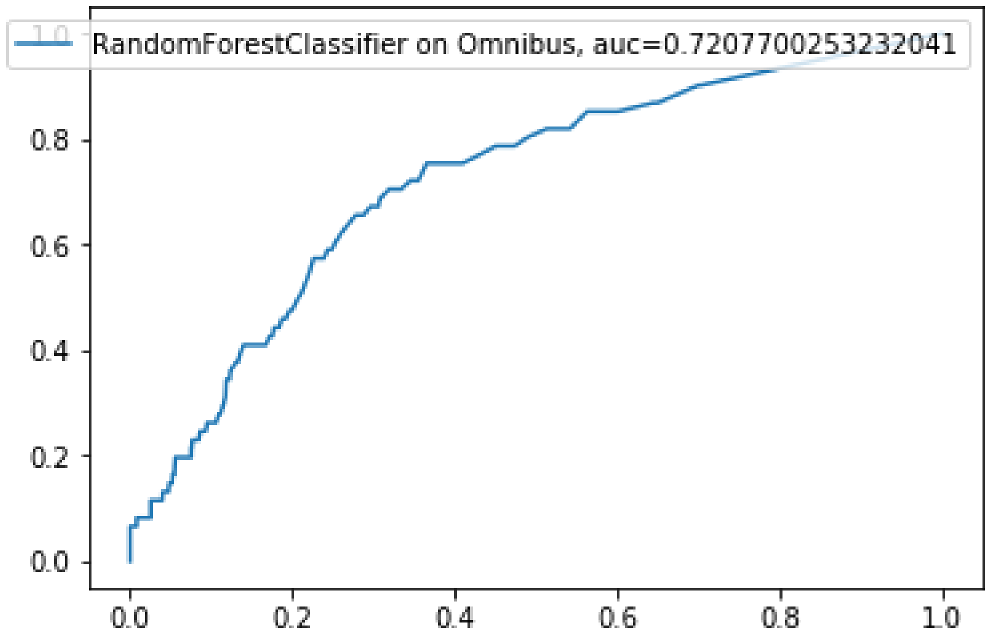


Figure 4.15: ROC-curve for RFC on gitlab-omnibus. Clear improvement from baseline

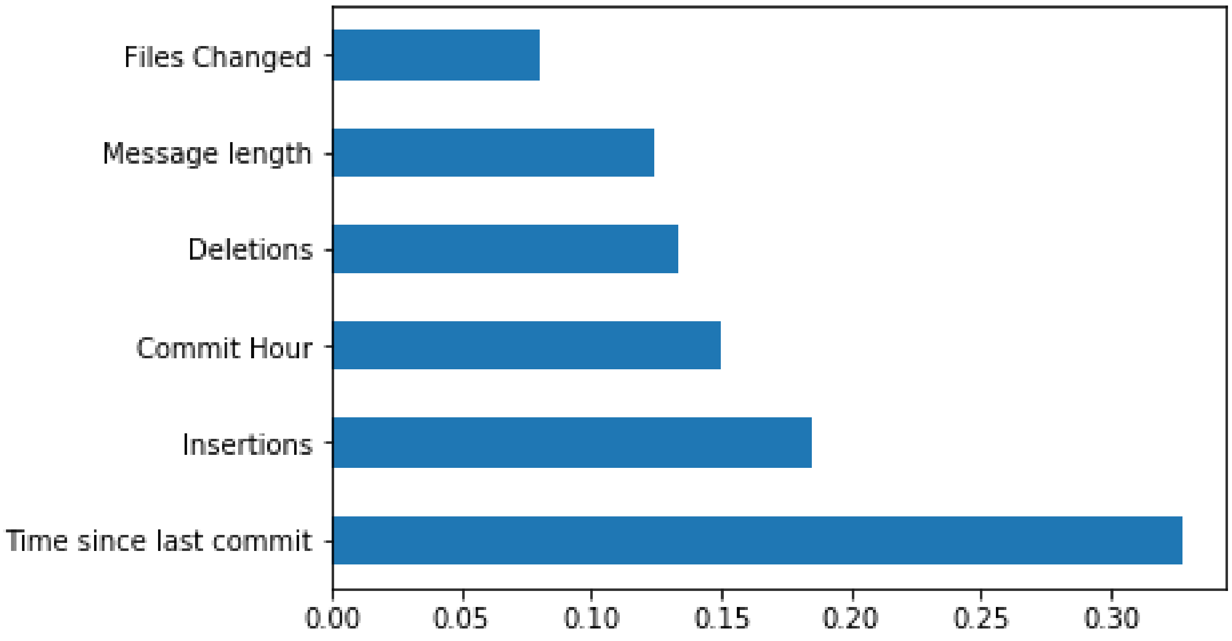


Figure 4.16: Feature importance for RFC on gitlab-omnibus ranked from lowest impact to highest

4.3.3 GitLab Kanban

Gitlab kanban is a small-scale (450 commits) open-source project. It is a tool that facilitates the usage of the kanban method.

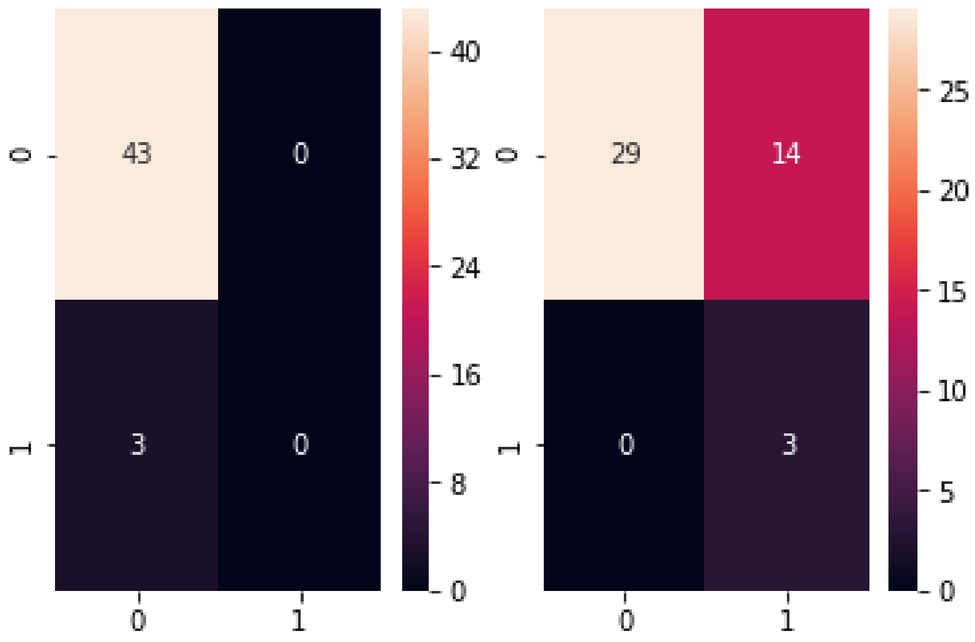


Figure 4.17: Confusion matrices for RFC on kanban. Worth noting is the small sample size. Naive model predicts all negative. Tuned one finds all the true positives.

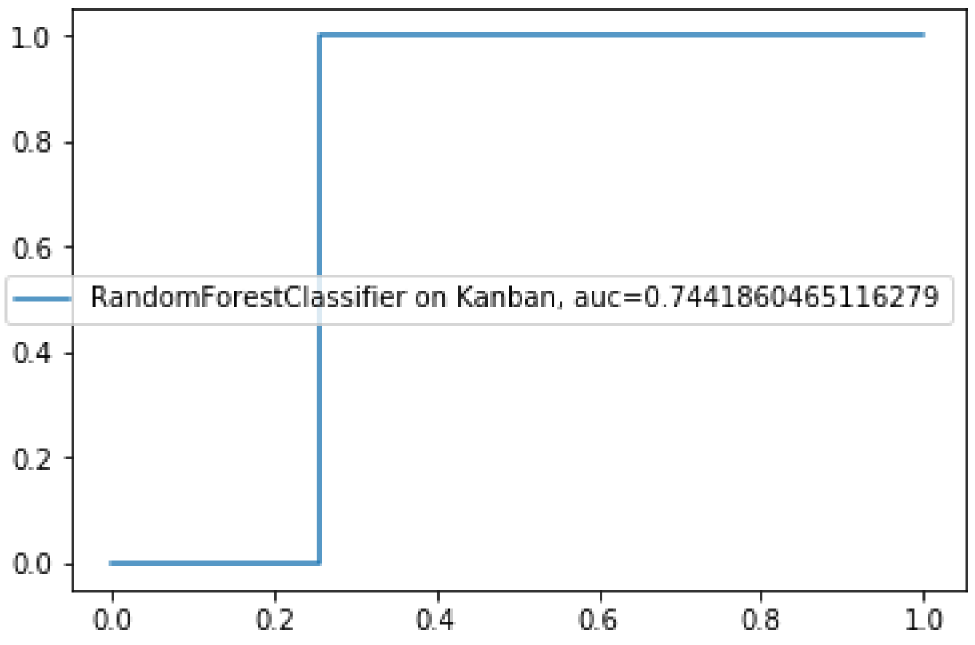


Figure 4.18: ROC-curve for RFC on kanban. The small sample size leads to a clear threshold.

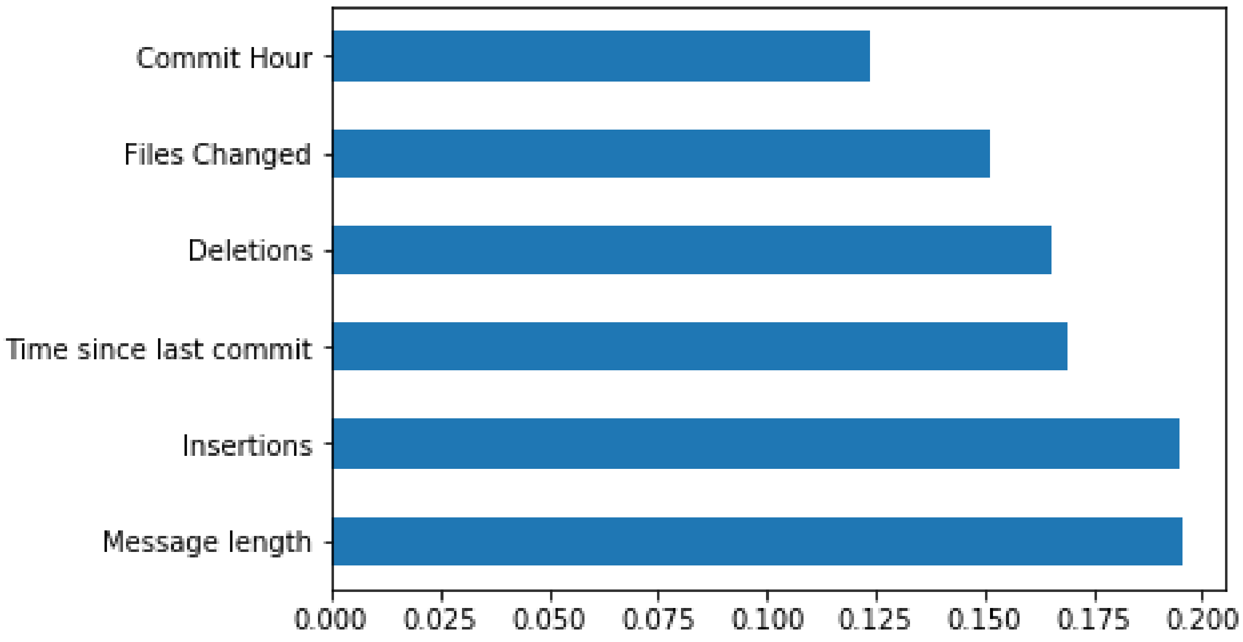


Figure 4.19: Feature importance of the RFC trained on kanban, ranked least impact to most.

Chapter 5
Findings

In this chapter we analyze the results, discuss them and the work as a whole, and try to reach final conclusions. Did we manage to answer the research questions?

5.1 Analysis

Here, the results from chapter 4 will be analyzed. The most important question is whether there is a significant correlation between the risk factors and the flagged faulty states, and if so, how strong is it? We will begin by studying the results from the manual statistical analysis. Subsequently, the results from the machine learning models trained on MSL will be analyzed. Lastly, the performance of the random forest classifier model on other select projects will be examined.

5.1.1 Implications from Manual Statistical Analysis

Looking at the results from Section 4.1, we see that all analyzed factors have a higher average for faulty commits, compared to non-faulty. The median is also higher for 4 out of 6 factors. This is a great indication that these factors have an impact on the bug risk. The student-t test further supports this, as all factors except for insertions and deletions can with 95% confidence be said to have a mean that is higher for faulty commits. However, simply the fact that the means are different does not necessarily say much about a factor's ability to predict the probability of failure. For example, the mean commit hour for non-faulty commits is 12,67 and 12,97 for faulty. Even though these proved to be statistically significantly different from each other, the difference is so slight (12:58 instead of 12:40) that it would not be possible to determine the risk of a commit solely based on this, especially when we have a standard deviation of 2,895 and 2,796 respectively. The other factors have a larger difference between their faulty and non-faulty commit means, varying from a 32% to a 48% larger value for the faulty commits. Intuitively, this would provide a stronger basis for prediction, but the large standard deviation causes issues for the prediction of these factors as well.

For example, the factor ”deletions” has the largest difference at 48,9% but the standard deviation of 1946 and 2115 (for means 176 and 262,1) causes uncertainty to whether the means can be said to differ at all. In conclusion, the manual statistical analysis indicates that there is a correlation between all factors and the bug risk. For commit hour, files changed, time since last commit, and message length, we can with 95% confidence say that faulty commits on average have a higher value. For insertions and deletions, we can say this with 77% and 74% confidence respectively. Because of the high variance of the data, it is difficult to construct a risk assessment model that adequately predicts the bug risk based on manual analysis.

5.1.2 Implications from Machine Learning

A machine learning approach turned out to be well suited for producing a predictor model. In this section, the results from Section 4.2 and 4.3 are examined and expanded upon.

Model Performance

Four different types of models were tested, and as the results show, the random forest classifier was by far the best. Naive Bayes and logistical regression both have an AUC-score of slightly above 50%, which means they are only marginally better than guessing. Support vector reaches an AUC-score of 0.6, which is a significant improvement over both baseline, and Naive Bayes and Logistical Regression, however, 0.6 is still generally considered poor. The random forest classifier reached a score of 0.76, which is definitely acceptable according to the rating presented in section 2.5.2.

Feature Importance

For MSL, i.e. the Sinch project, we see that time since last commit is the biggest indicator of bug risk, followed by insertions and deletions, these results are discussed further in Section 5.2.1. Also, important to notice is that none of the six factors had no importance. From the generalized testing it is apparent that the method holds water even on other projects. We have actively tried to test on different sized projects, all with successful results. Worth noting is that the results have in no way been cherry-picked. From the respective feature importance graphs, we see that the biggest risk indicators vary from project to project. Table 5.1 ranks the factors based on their average feature importance from the four projects, from most important to least important. The averages have been weighted equally, independent on project size.

Factor	Average Feature Importance
Time since last commit	0.218
Insertions	0.208
Deletions	0.167
Message length	0.154
Commit hour	0.135
Files changed	0.118

Table 5.1: Average feature importance for all four projects, ranked from most important to least important, based on the ML results

5.2 Discussion

In this section, the project will be discussed in three areas:

- **General Discussion**, where takeaways and considerations from throughout the thesis are discussed.
- **Limitations**, where the tool is examined from the perspective of what it cannot do.
- **Future Works**, where we outline likely paths to circumvent limitations or to improve the performance of the tool even further.

Some points are reiterations and summaries of discussion already sprinkled through the work, some points are reflections and considerations that were had but didn't fit in any of the other sections.

5.2.1 General Discussion

Factor Significance

Some noteworthy results from Section 4 are that insertions and deletions were the only factors not statistically significant at a 95% confidence level. Hypothesised beforehand was that these two factors would be the most prominent predictors of the bug risk. Although, if we were to only look at the average values, these two factors had the largest percentual differences between non-faulty and faulty commits. As discussed previously, it is the large variation that causes uncertainty and it could very well still be that insertions and deletions are the biggest indicators of bug risk, but there is not sufficient proof from the statistical analysis that this is the case. However, when looking at the feature importance obtained from machine learning, we see that insertions and deletions are ranked as number 2 and 3 in terms of their average ability to predict a faulty commit. This provides some backing to them actually being significant indicators of the bug risk. In addition to this, insertions and the time since last commit are the only factors that have a feature importance above 0.15 for all projects, with time since last commit making a strong case for being the most important factor, with an average score of 0.21 and a maximum of above 0.3.

A big surprise from the ML results was that number of files changed was ranked last. It is reasonable to believe that changing a large number of files would cause the risk to increase, and logically it should be correlated quite heavily with the number of insertions and deletions, which seemingly have a large impact. It is though entirely possible that many changes in few files would be riskier than a few changes in many files, but it is not possible to draw any solid conclusions regarding this. When looking at forums like stack overflow, some argue that when there is a strong correlation between factors, the resulting feature importance from a random forest model can be lower. If this is true, it can be an explanation to the differences seen in files changed and insertions, but for now, this is only speculation.

Counter-productivity Through "Playing" the Model

An implementational aspect that was initially thought to be a problem was that the developers could learn what causes a commit to get a high risk score, and adapt to this. The most obvious example would be to consciously write shorter commit messages, which would not lower the actual commit risk but only lower the risk score, as well as cause worse and less informative commit messages. However, since the model is dynamic and continuously trained on the data from the same project that is being developed, the model would in turn learn this changed behaviour and understand that commit message length no longer would be a good indicator of the commit risk. The dynamic nature of the tool can be enforced even further through weighing more recent results heavier or through only looking at commits within a defined time-frame. It is also worth noting that it is hoped that the developers will feel that the tool is providing assistance rather than being a nuisance, which would discourage this kind of behaviour altogether.

Generalizeable Results

One interesting takeaway is that the model achieved decent to good performance on all the tested projects, ranging from small-scale to large-scale. This suggests that the method extends well, and could be applied to many use cases at companies.

5.2.2 Limitations

The results of the research and tool design are satisfactory, and believed to be of some value to Sinch AB. However, our approach has its limitations. These, and proposed solutions where applicable, are listed in this section.

Faulty State and Bug-Inducing Commit Approximation

One major limitation that needs to be considered is the labeling of faulty commits. The entire study is based on this approximation and that it can give an adequate estimate of which commits caused a bug and which did not. If the assumption were to be wrong, the results achieved would lack any practical meaning as it only predicts what we have defined as bugs. In order to validate whether this labeling approach is sufficient, similar tests with different labeling approaches could be performed. To manually label the commits would likely be a very good validation technique, although this would be incredibly time demanding and tedious, making it fall out of the scope of this project, and instead a subject for future work.

Another consideration related to the labeling of faulty commits, is that it also requires the development team to practice good commit hygiene, at least in terms of always writing a message containing something along the lines of ”bug” or ”revert” when fixing a bug. Preferably, the team should decide on a label to include in the commit message when fixing a bug, that is unique for the act of bug fixing, and unlikely to be included in an unrelated commit message. An example could be ”BFX” or ”BGFX”. If this is overlooked, the entire labeling process runs the risk of being incorrect and either not labeling enough or too much and incorrectly, causing the results to be skewed. One workaround could be basing the bug spotting on issues in JIRA marked with bug, this will be discussed further in section 5.2.3.

High Data Variance

In an optimal world, the model would perform with a close to 100% precision. The data, however, is inconclusive in nature. A commit that changes one line in one file can potentially crash the whole system, while an all-encompassing change may not cause any issues at all. This high variance makes it difficult to predict with certainty. It is highly probable that taking more factors into consideration would improve the model's performance, but this effect will experience diminishing returns, and likely never get to a point where there will be no edge cases.

Lack of Data in Smaller Projects

Our method studies historical data, in order to predict the future. In small scale projects, this historical data might be insufficient to find the correlations necessary for valid predictions. In our case, the small scale Kanban-project was enough to have a decent result. However, we can see from the ROC-curve that its predictions are blunt, thus rendering a good classifier, but poor probability predictor.

5.2.3 Future works

The initial scope of the project was vast, and our results suggest that the area seems to have a lot of potential. As such, there are areas that the time frame did not allow to be explored properly, but that its hypothesized would improve the performance even further. These areas will be discussed in this section.

Pipeline Integration

This paper has been the groundwork and proof of concept for a CI/CD compatible risk assessment tool. To integrate it into the CI/CD flow at the company, it is believed that we need to work closely with the end-users. In order for people to actually use our tool, it needs to be easy to install and use, and present results that users can appreciate the value of instantly. The easiest way of solving this would be packaging it as a GitLab component. This allows users to easily integrate the service into the designated location in the pipeline.

Commit Hygiene Framework

It is possible that an even higher model performance might be achievable if a framework for commit hygiene is put into place. For example, tagging commits as refactors, merges, bugfixes, etc. might allow for another dimension of statistics. This, however, has some adverse implications. Getting a whole company to start working differently is a vast undertaking, and there's no guarantee that the approach will ever be adapted to an extent where it is useful. The method would also reintroduce the human factor to the risk assessment tool, something that is undesirable.

More factors

We did not have time to fully study every factor in the E/R diagram mocked up in section 3.2. However, it is believed that taking these factors into account would both increase the precision of the model, and perhaps even help to approximate the impact. The author, and project scopes are possible to construct with information that is available to us right now, and our tool is built with this extension in mind. The code quality metrics scope depends on there being a static code analyzing tool implemented company-wide that can dump machine-readable stats about the code in question, and has an API that can be called.

Deep Learning

Machine learning is an area that allows for vast experimentation. One can try any number of models, tune hyper-parameters, utilize different ways of data normalization, and all of these factors might impact the performance of the model. It is possible that when for example including more factors, that the performance would be increased by utilizing a deeper network.

Faulty State Approximation

As discussed in the last section. One of the biggest limitations of our tool is that its predictions are based solely on the approximations of what commits have caused a faulty state (the method described in section 3.2.3). Hence, the predictions will never be better than the approximation. Our results make it likely that it is a valid approximation, but there might be other, better ways of spotting adverse events. One of the ways that exist, but might be unfeasible is manual traversing of the commits, and interviews with stakeholders that remember when things have gone awry, and what caused it. Basing the analysis on Jira issues is also a likely way forward. Furthermore, letting the program differ between the four types of bugs shown in figure 2.4 could also lead to more representative data.

5.3 Conclusions

Here we tie back to the research questions. Are the results conclusive enough to have scientifically backed answers to both of them?

RQ1: Are there development-related meta factors that can be extracted from common versioning tools, that can help in assessing risk associated with a deploy or commit?

In the analysis, in Section 5.1, we see that an acceptable prediction of the bug risk was achieved, something that given the inconclusive nature of the data has to be considered a good result. However, due to the grand scope of the research, some limitations had to be imposed. First and foremost, the real-world prediction will never be better than our approximation of what constitutes a bug, outlined in Section 3.2.3. How well the results fit with our hypotheses, and scientific backing in literature regarding the SZZ-algorithm supports that it is a valid approximation, but this could be tested further for a truly conclusive answer. Ways of doing this are outlined in Section 5.2.3. The service is also heavily reliant on the project being git-based. However, these days, git is so ubiquitous that it fulfills the common versioning tools sub-requirement of question R1. With this context in mind, the results from the machine learning show that there's a clear correlation between some factors that are central, and general to all projects based in a git environment, and between faulty states. Furthermore, these correlations can be estimated by machine learning models. Taking into account our research about risk, and the results from the case study, it is believed that these predictions help greatly in risk assessment. As such we are rather confident that the answer to the question is yes.

RQ2: Can this risk assessment method be utilized to implement an automated last-line-of-defence service integrated into the currently used development pipeline, and how would this benefit Sinch?

Our research on DevOps in Section 2.1, and the case study at Sinch, especially Section 3.2, made it clear that a tool that could help in assessing the Implementation Complexity of a deployment, is a tool that is valuable. The answer to question R1 shows that this assessment is possible. Furthermore, the limitations we have imposed on ourselves, such as everything needing to be automated and lightweight, makes it a good fit for integration in a DevOps pipeline. In order to fully explore this question, there would need to be a trial in production-environment at Sinch. However, based on the stakeholder's responses we are fairly confident that the tool can be integrated into their toolkit, and that this would bring substantial value, especially considering the rudimentary nature of the assessment process currently employed. Just having some data to base the review process is believed to be a big step forward. The fact that the service works best at predicting probability of risk, rather than impact is also important, since this is the area which left most to wish for in the current process. One more concrete area where the tool is believed to help, is in achieving ISO-27001 certification, which requires the deployment risk management to involve automated gauntlets through which all code headed for production must pass. This is exactly the type of use-case that our research indicates our service would perform well on.

Summary

In general, we are happy with the results and feel that there is a multitude of cases where the method could be utilized to facilitate risk management work at all levels. We are also certain that there are further considerations that might make the tool even more sensitive and accurate, and hope that it can act as an important piece of the puzzle for the step into truly data-driven decision making.

References

[1] What is machine learning? https://www.technologyreview.com/2018/11/17/103781/what-is-machine-learning-we-drew-you-another-flowchart/, 2017. [Online; accessed 2020-05-27].

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[3] Amazon. What is DevOps? https://aws.amazon.com/devops/what-is-devops/, 2020. [Online; accessed 25-March-2020].

[4] Henri Barki, Suzanne Rivard, and Jean Talbot. Toward an assessment of software development risk. *Journal of management information systems*, 10(2):203–225, 1993.

[5] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, 1994.

[6] Barry Boehm and Richard Turner. Management challenges to implementing agile processes in traditional development organizations. *IEEE software*, 22(5):30–39, 2005.

[7] Barry W. Boehm. Software risk management: principles and practices. *IEEE software*, 8(1):32–41, 1991.

[8] Joe Brady. Risk assessment: Issues and challenges. *Pharmaceutical Engineering*, 2015.

[9] Lionel C Briand, Jurgen Wust, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 475–482. IEEE, 1999.

[10] Jason Brownlee. How To Know if Your Machine Learning Model Has Good Performance. https://aws.amazon.com/devops/what-is-devops/, 2018. [Online; accessed 23-May-2020].

[11] Cagatay Catal. Performance evaluation metrics for software fault prediction studies. *Acta Polytechnica Hungarica*, 9(4):193–206, 2012.

[12] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[13] CodeMR. Codemr is an architectural software quality and static code analysis tool., 2020.

[14] Dronasschool. Devops workflow, 2018.

[15] Marlon Dumas, Wil M Van der Aalst, and Arthur H Ter Hofstede. *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, 2005.

[16] Martin S Feather. Towards a unified approach to the representation of, and reasoning with, probabilistic risk information about software and its system interface. In *15th International Symposium on Software Reliability Engineering*, pages 391–402. IEEE, 2004.

[17] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[18] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 1–9, 2014.

[19] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[20] Say-Wei Foo and Arumugam Muruganantham. Software risk assessment model. In *Proceedings of the 2000 IEEE International Conference on Management of Innovation and Technology. ICMIT 2000.'Management in the 21st Century'(Cat. No. 00EX457)*, volume 2, pages 536–544. IEEE, 2000.

[21] Martin Fowler and Matthew Foemmel. Continuous Integration. `https://martinfowler.com/articles/originalContinuousIntegration.html`, 2006. [Online; accessed 16-June-2020].

[22] Mithat Gönen et al. Receiver operating characteristic (roc) curves. *SAS Users Group International (SUGI)*, 31:210–231, 2006.

[23] Daya Gupta and Mohd Sadiq. Software risk assessment and estimation model. In *2008 International Conference on Computer Science and Information Technology*, pages 963–967. IEEE, 2008.

[24] Sean Guthrie. How to Eliminate Silos Using DevOps. `https://orangematter.solarwinds.com/2018/08/09/how-to-eliminate-silos-using-devops/`, 2017. [Online; accessed 2020-03-10].

[25] Sean Guthrie. DevOps Principles- The CAMS Model. `https://medium.com/@seanguthrie/devops-principles-the-cams-model-9687591ca37a`, 2018. [Online; accessed 2020-03-06].

[26] Henry Hinnefeld. Predicting code bug risk with git metadata. `https://medium.com/civis-analytics/predicting-code-bug-risk-with-git-metadata-bc0b675ad28f`.

[27] Glyn A Holton. Defining risk. *Financial analysts journal*, 60(6):19–25, 2004.

[28] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[29] Jez Humble and Joanne Molesky. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, 24(8):6, 2011.

[30] Michael Hüttermann. *DevOps for developers*. Apress, 2012.

[31] Martin Höst. Edaa35 utvärdering av programvarusystem vt 2020. 2020.

[32] Cygnet Infotech. DevOps Bringing Paradigm Shift to Software Development LifeCycle Management. `https://www.cygnet-infotech.com/blog/devops-bringing-paradigm-shift-to-software-development-lifecycle-management`, 2017. [Online; accessed 2020-05-27].

[33] Joseph Ingeno. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018.

[34] Seyed Hossein Iranmanesh, Seyed Behrouz Khodadadi, and Shakib Taheri. Risk assessment of software projects using fuzzy inference system. In *2009 International Conference on Computers & Industrial Engineering*, pages 1149–1154. IEEE, 2009.

[35] Mira Kajko-Mattsson and Jaana Nyfjord. State of software risk management practice. *IAENG international journal of Computer Science*, 35(4), 2008.

[36] Mark Keil, Paul E Cule, Kalle Lyytinen, and Roy C Schmidt. A framework for identifying software project risks. *Communications of the ACM*, 41(11):76–83, 1998.

[37] Mik Kersten. A cambrian explosion of devops tools. *IEEE Annals of the History of Computing*, (02):14–17, 2018.

[38] David Linthicum. DevOps automation best practices: How much is too much? `https://techbeacon.com/devops/devops-automation-best-practices-how-much-too-much`, 2018. [Online; accessed 05-May-2020].

[39] Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen. *Model-driven risk analysis: the CORAS approach*. Springer Science & Business Media, 2010.

[40] Medium. Train and test set in python machine learning — how to split, 2020.

[41] Stefan Metzger, David Durden, Cove Sturtevant, Hongyan Luo, Natchaya Pingintha-Durden, Torsten Sachs, Andrei Serafimovich, Jörg Hartmann, Jiahong Li, Ke Xu, et al. eddy4r 0.2. 0: a devops model for community-extensible processing and analysis of eddy-covariance data based on r, git, docker, and hdf5. *Geoscientific Model Development*, 10:3189–3206, 2017.

[42] Tony Moynihan. How experienced project managers assess risk. *IEEE software*, 14(3):35–41, 1997.

[43] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[44] Sarang Narkhede. Understanding auc-roc curve. *Towards Data Science*, 26, 2018.

[45] Simone Natale and Andrea Ballatore. Imagining the thinking machine: Technological myths and the rise of artificial intelligence. *Convergence*, 26(1):3–18, 2020.

[46] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. Towards a classification of bugs to facilitate software maintainability tasks. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, pages 25–32, 2018.

[47] C.R. Parikh and H. [Thiessen Philbrook]. Chapter two - statistical considerations in analysis and interpretation of biomarker studies. In Charles L. Edelstein, editor, *Biomarkers of Kidney Disease (Second Edition)*, pages 21 – 32. Academic Press, second edition edition, 2017.

[48] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Sklearn: Machine learning in python. *Journal of machine learning research*, 2011.

[49] Grant Purdy. Iso 31000: 2009—setting a new standard for risk management. *Risk Analysis: An International Journal*, 30(6):881–886, 2010.

[50] Zornitza Racheva, Maya Daneva, and Klaas Sikkel. Value creation by agile projects: Methodology or mystery? In *International Conference on Product-Focused Software Process Improvement*, pages 141–155. Springer, 2009.

[51] Gema Rodriguez, Gregorio Robles, and Jesus Gonzalez-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 03 2018.

[52] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M González-Barahona. What if a bug has a different origin? making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–4, 2018.

[53] Janne Ropponen and Kalle Lyytinen. Components of software development risk: How to address them? a project manager survey. *IEEE transactions on software engineering*, 26(2):98–112, 2000.

[54] Ronald S Ross. Guide for conducting risk assessments. Technical report, 2012.

[55] Margaret Rouse. What is Machine Learning. `https://searchenterpriseai.techtarget.com/definition/machine-learning-ML`, 2020. [Online; accessed 16-June-2020].

[56] Mohd Sadiq, Md Khalid Imam Rahmani, Mohd Wazih Ahmad, and Sher Jung. Software risk assessment and evaluation process (sraep) using model based approach. In *2010 International Conference on Networking and Information Technology*, pages 171–177. IEEE, 2010.

[57] Roy Schmidt, Kalle Lyytinen, Mark Keil, and Paul Cule. Identifying software project risks: An international delphi study. *Journal of management information systems*, 17(4):5–36, 2001.

[58] Atlassian Sten Pittet. Continuous integration vs. continuous delivery vs. continuous deployment. `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment`, 2020. [Online; accessed 05-May-2020].

[59] Megan Sumrell. From waterfall to agile-how does a qa team transition. In *Proceedings of the Agile Conference (AGILE)*, pages 291–295, 2007.

[60] Thomas G. Tape. Interpreting Diagnostic Tests. `http://gim.unmc.edu/dxtests/roc3.html`, 2020. [Online; accessed 16-June-2020].

[61] Omni Thakur and Janpreet Singh. A review study: automated risk identification tool for software development process. *Orient J Comp Sci Technol*, 2014:167–172, 2014.

[62] Massood Towhidnejad, Li Shen, and Thomas B Hilburn. Application of software fault tree analysis to an airport ground control system. In *Software engineering research and practice*, pages 67–71, 2008.

[63] David P. Voorhees. *Characteristics of Good Software Design*, pages 127–136. Springer International Publishing, Cham, 2020.

[64] Linda Wallace, Mark Keil, and Arun Rai. How software project risk affects project performance: An investigation of the dimensions of risk and an exploratory model. *Decision sciences*, 35(2):289–321, 2004.

[65] Juliano Araujo Wickboldt, Luís Armando Bianchin, Roben Castagna Lunardi, Fabrício Girardi Andreis, Weverton Luis da Costa Cordeiro, Cristiano Bonato Both, Lisandro Zambenedetti Granville, Luciano Paschoal Gaspary, David Trastour, and Claudio Bartolini. Improving it change management processes with automated risk assessment. In *International Workshop on Distributed Systems: Operations and Management*, pages 71–84. Springer, 2009.

[66] Edward Yourdon and Larry L Constantine. *Structured design: fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979.

Appendices

Appendix A
Factors for Risk Assessment

A.1 Identifying Software Project Risks: An International Delphi Study [57]

1. Lack of top management commitment to the project
2. Failure to gain user commitment
3. Misunderstanding the requirements
4. Lack of adequate user involvement
5. Lack of required knowledge/skills in the project personnel
6. Lack of frozen requirements
7. Changing scope/objectives
8. Introduction of new technology
9. Failure to manage end user expectations
10. Insufficient/inappropriate staffing
11. Conflict between user departments
12. Lack of cooperation from users
13. Change in ownership or senior management
14. Staffing volatility

15. Lack of effective development process/methodology
16. Not managing change properly
17. Lack of effective project management skills
18. Lack of effective project management methodology
19. Unclear/misunderstood scope/objectives
20. Improper definition of roles and responsibilities
21. Number of organizational units involved
22. No planning or inadequate planning
23. Artificial deadlines
24. Multi-vendor projects complicate dependencies
25. Lack of ”people skills” in project leadership
26. Trying new development method/technology during important project
27. Bad estimation
28. New and/or unfamiliar subject matter for both users and developers
29. Poor or nonexistent control

A.2 Risk Assessment of Software Projects Using Fuzzy Inference System [34]

1. Corporate Environment
2. Sponsorship/Ownership
3. Relationship Management
4. Project Management
5. Scope
6. Requirements
7. Funding
8. Scheduling Planning
9. Development Process
10. Personnel Staffing
11. Technology
12. External Dependencies

A.3 Software Risk Management: Principles and Practices [7]

1. Personnel shortfalls
2. Unrealistic schedules and budgets
3. Developing the wrong functions and properties
4. Developing the wrong user interface
5. Gold-plating
6. Continuing stream of requirements changes
7. Shortfalls in externally furnished components
8. Shortfalls in externally performed tasks
9. Real-time performance shortfalls
10. Straining computer-science capabilities

A.4 Components of Software Development Risk: How to Address Them? A Project Manager Survey [53]

1. Scheduling and timing risks.
2. System functionality risks.
3. Subcontracting risks.
4. Requirement management risks.
5. Resource usage and performance risks.
6. Personnel management risks.

A.5 How Experienced Project Managers Assess Risk [42]

1. he client's knowledge/understanding/clarity regarding 12 14 the requirements/problem to be solved
2. The existence/competence/seniority/commitment of the project patron/owner

3. Level of IT competence and experience of the customer/users
4. Need to integrate/interface with other systems
5. Scale/coordination complexity of the project (numbers of disciplines, need to share resources, need to subcontract, and so on)
6. Main source of control over the project (developer versus client versus third parties)
7. Level of change to be experienced by the client (to procedures, workflow, structures, and so on)
8. The need to satisfy multiple groups of disparate users versus the need to satisfy one group of similar users
9. Who we will be working through: users versus the IT department, individuals versus committees
10. Developer's familiarity with platform/environment/methods
11. Developer's previous experience with the application
12. Level of enthusiasm/support/"energy" for the project in the client's organization
13. Logical complexity of the application
14. Ease of solution validation (for example, possibility of prototyping)
15. Client's willingness/capability to handle implementation
16. Freedom of choice of platform/development environment
17. Criticality/reversibility of the new system roll-out
18. Maturity of the technology to be used
19. Developer's knowledge of country/culture/language
20. Stability of the client's business environment
21. Developer's knowledge of client's business sector
22. Other constructs (hard to classify)

A.6 Toward an Assessment of Software Development Risk [4]

1. Size
2. Team size

3. Technical complexity
4. External suppliers
5. Technical newness
6. Application newness
7. Resource sufficiency
8. Task characteristics
9. Team diversity
10. Team expertise (software development)
11. Team expertise (application)
12. Team expertise (task)
13. Team expertise (in general)
14. Experience of leader
15. Number of users
16. Diversity of users
17. User attitudes
18. Conflicts
19. Top management support

A.7 A Review Study: Automated Risk Identification Tool for Software Development Process [61]

- **Risk in software requirements**
  1. No proper requirements gathering.
  2. No proper documentation.
  3. The definition of the requirement is very poor.
  4. On the spot change of requirements.
- **Risk in software cost**
  1. Poor estimation of the projects cost.
  2. Poor working of the hardware.

3. No proper testing techniques used.
4. No proper monitoring.
5. Due to the Complexity of architecture.
6. Also due to the large size of architecture.

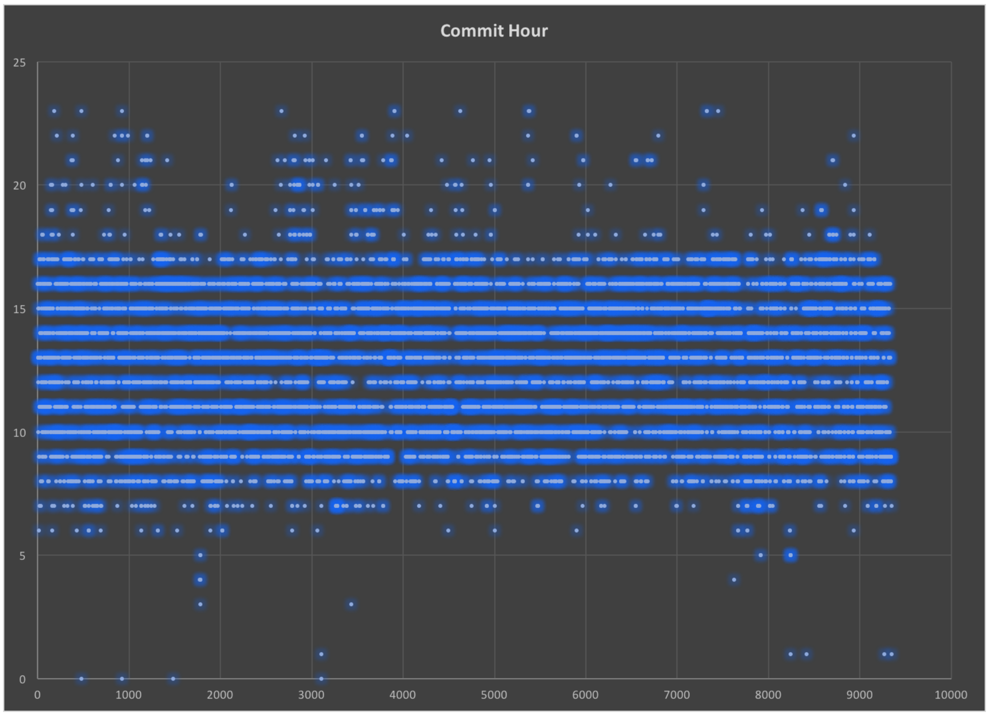
- **Software Scheduling Risk**

  1. Due to the change in requirements and their extension.
  2. Inadequate knowledge about tools and techniques.
  3. Due to the lack of manager experience.
  4. Due to the lack of knowledge and the skills.
  5. Due to the lack of good estimation in projects.

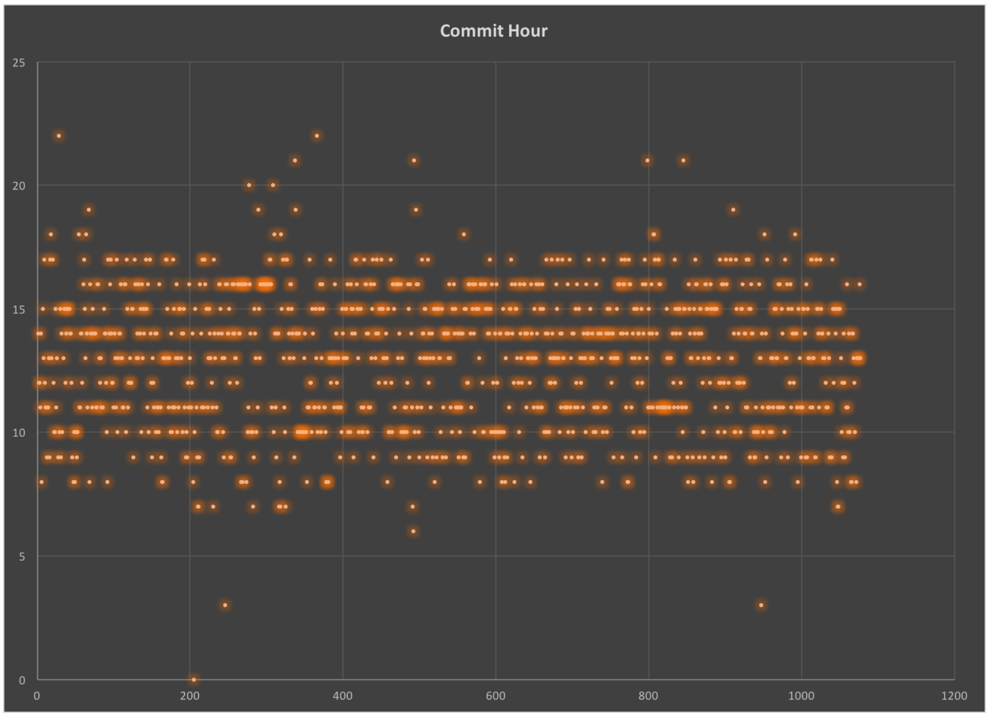
- **Software Quality Risk**

  1. Due to the extension of requirements change.
  2. Absence of design documentation.
  3. Due to the inadequate documentation.
  4. Due to the absence of project standard.
  5. No proper estimation of budget.
  6. Poor definition of requirements.
  7. Due to insufficient knowledge skills.

Appendix B
Scatter Plots

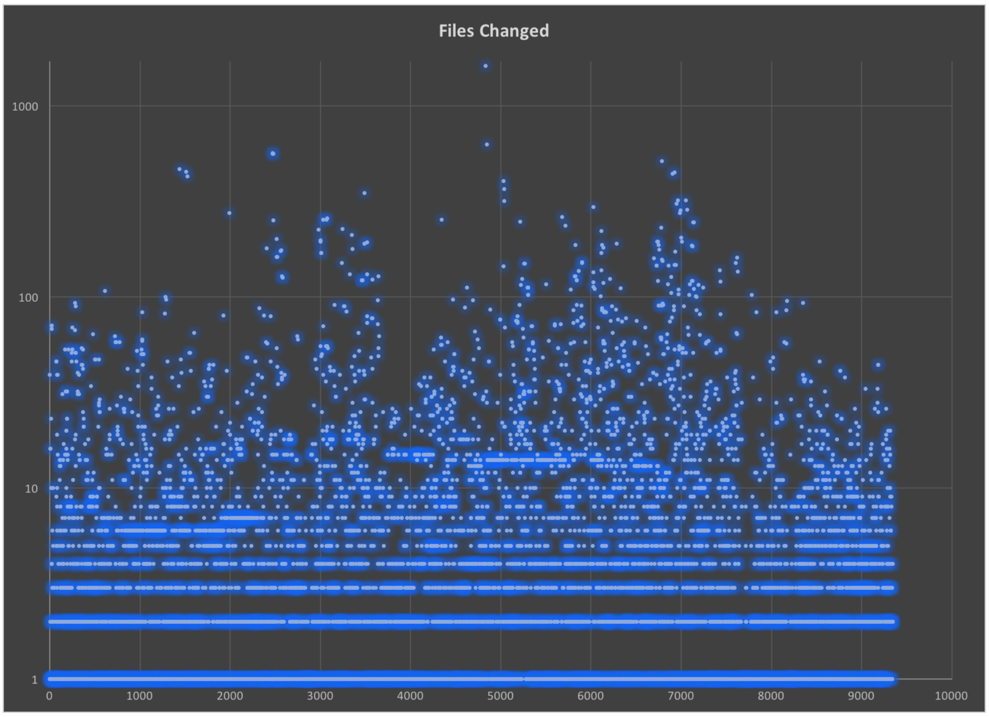


(a) faulty

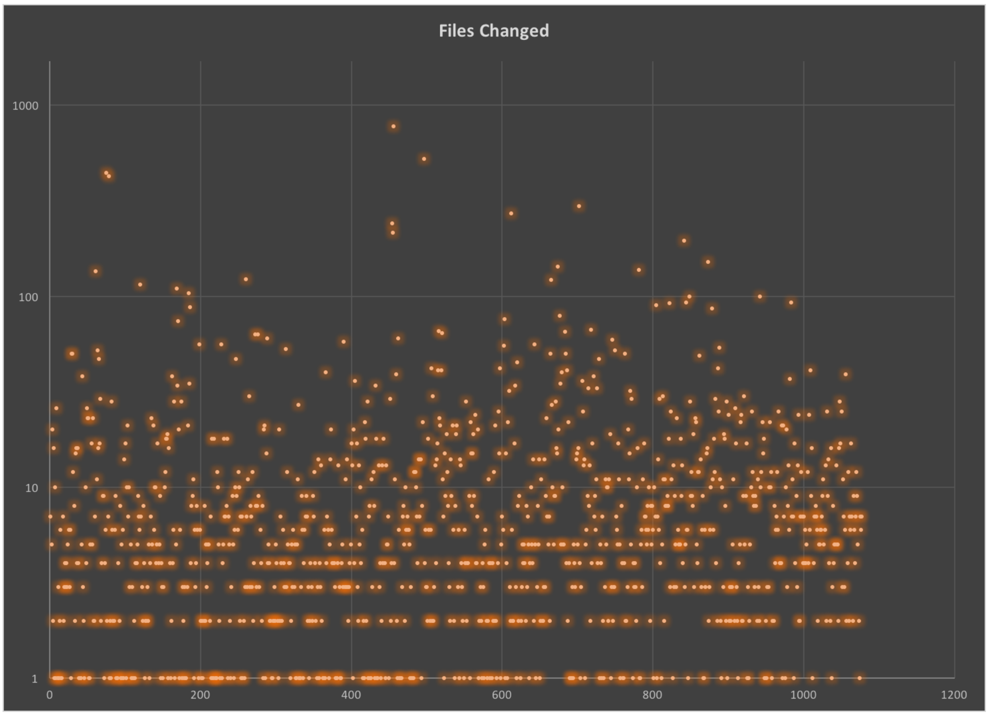


(b) non-faulty

Figure B.1: faulty and non-faulty scatter plots for commit hour.

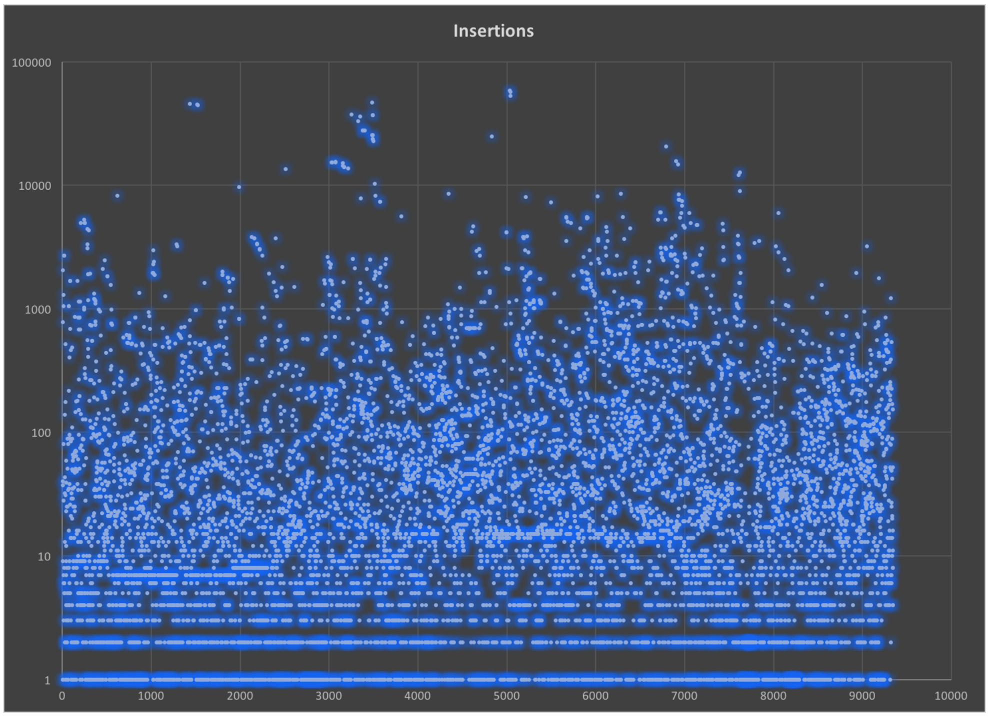


(a) faulty1

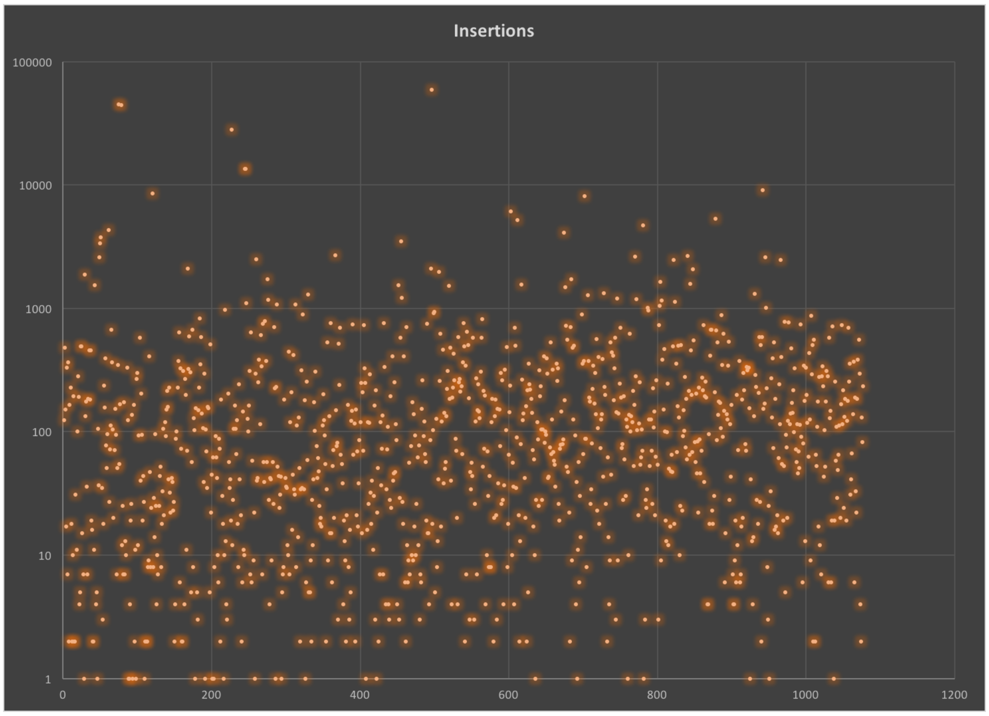


(b) non-faulty

Figure B.2: faulty and non-faulty scatter plots for files changed.

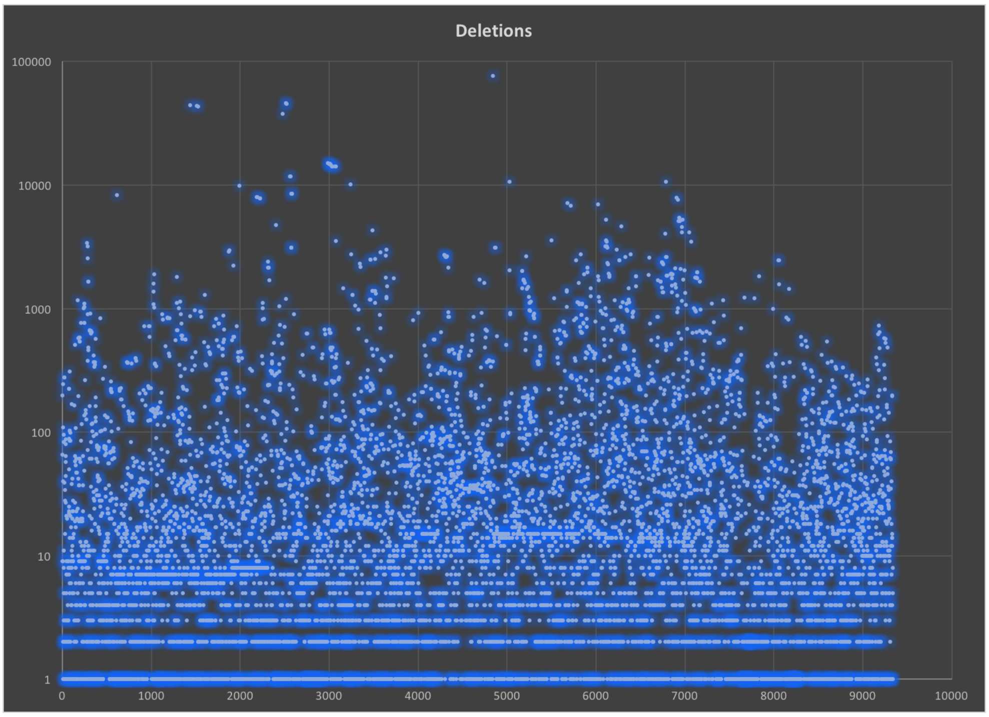


(a) faulty

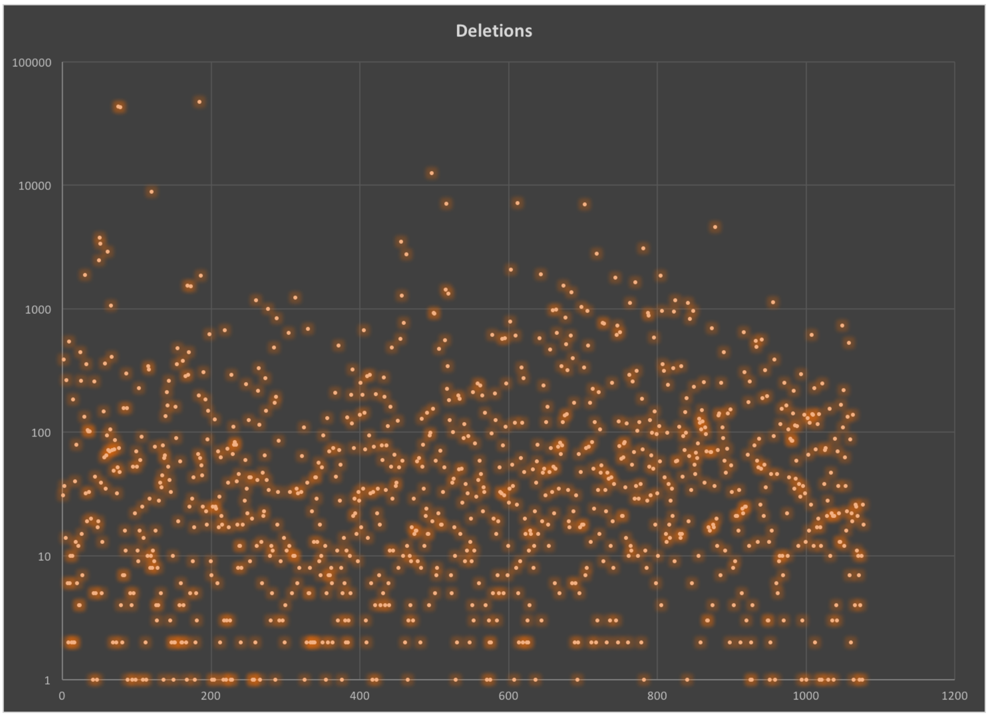


(b) non-faulty

Figure B.3: faulty and non-faulty scatter plots for insertions.

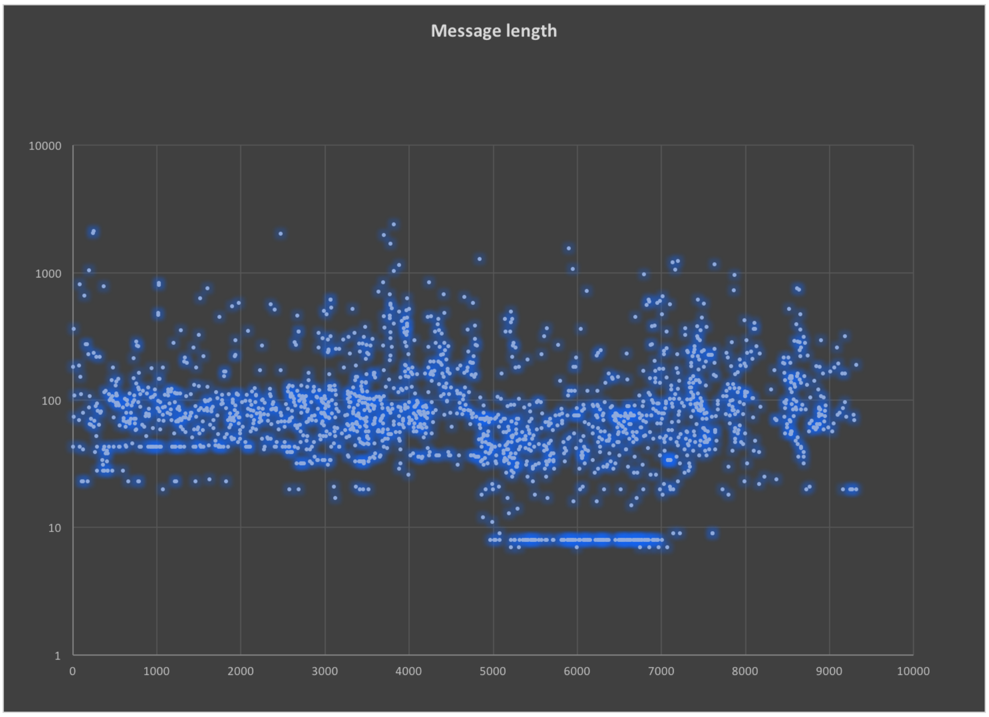


(a) faulty

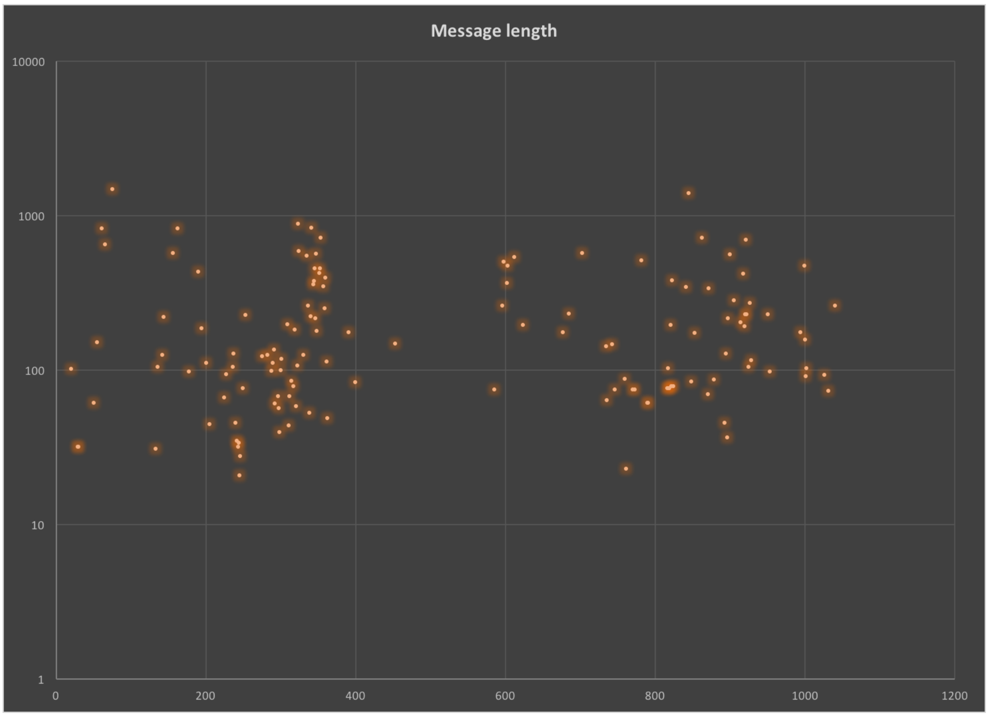


(b) non-faulty

Figure B.4: faulty and non-faulty scatter plots for deletions.



(a) faulty



(b) non-faulty

Figure B.5: faulty and non-faulty scatter plots for message length.

Exjobb: *Automated Fuzzy Logic Risk Assessment and its Role in a DevOps Workflow*
Examinator: Martin Höst | Handledare: Ulf Asklund | Studenter: Karl Lundberg och Andreas Warvsten
Institutionen för datavetenskap | LTH | Presenterad 2020-06-11

Datadriven Riskanalys ur ett DevOps-perspektiv

En populärvetenskaplig sammanfattning

Klimatet i dagens IT-sektor är sådant att man knappt kan bestämma sig för vilken kaffe man ska ha utan att först tillhandahålla data för att backa upp sitt beslut. Detta datadrivna paradigmskifte har slått rot i nästan alla delar av företagens aktiviteter och har fört med sig en uppsjö fördelar. Det finns dock ett område som vägrar låta sig datafieras - riskhantering. Till dags datum är riskhanteringsprocessen ofta löst baserad på allt ifrån subjektiv utvecklarkompetens till ren magkänsla. Denna artikel visar att denna process till viss del kan automatiseras, något som kan underlätta företags arbete oerhört.

För att kunna ta datadrivna beslut krävs data. Det är här ett annat nyligt paradigmskifte kommer in. **DevOps** är ett portmanteau av de engelska orden *Development* och *Operations*, och är ett arbetssätt som bl.a. lägger stor vikt vid att man sparar metainformation om utvecklingsprocessen. Ofta görs detta med hjälp av versioneringsverktyg såsom git och Jira. Dessa verktyg lagrar data om "*commits*", eller "*ändringar*". I vårt arbete var det ur denna datamängd de potentiella riskfaktorerna extraherades. Vår slutgiltiga lista över faktorer vi extraherade ur varje ändring är:

- Hur många olika filer som berörts
- Hur många rader som har lagts till
- Hur många rader som har tagits bort
- Tid sedan projektet ändrades senast
- Tiden på dagen då ändringen skedde
- Längden på meddelandet som beskrev ändringen

Med dessa faktorer extraherade var det dags för nästa steg. Riskhantering går ut på att minimera exponering till händelser med negativ påverkan. För att hitta det historiska sambandet mellan faktorerna och dessa negativa händelser utnyttjades en snillrik algoritm som går bakvägen från de ändringar som fixat kod. Tanken är att man utnyttjar dessa "*fixes*", för att hitta vilken kod som har behövts fixas. Sedan hittar man *ändringen* som gav upphov till dessa rader som behövde fixas - denna *ändring* blir sedan klassificerad som en källa till en negativ händelse.

Nu har vi en uppsättning riskfaktorer per ändring, och en approximering av om ändringen gav upphov till en negativ händelse eller inte. Allt som kvarstår att göra är att undersöka om det finns ett tillräckligt starkt samband för att kunna sia om ifall en kommande ändring medför stor eller liten risk. För att undersöka detta utnyttjades maskininlärning. Våra modeller visar att det finns ett tydligt samband mellan en *ändrings* faktorvärden och risk. Detta resultat kan användas som underlag för datadrivna beslut i samband med release av ny programvara och kod.