Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
Spring term 2022 | LIU-IDA/LITH-EX-A--2022/082--SE

Confidence in Release Candidates

– Maintaining confidence levels when moving from traditional release management to continuous delivery

---

Förtroende för releasekandidater - Bibehållande av förtroendenivåer vid byte från traditionell releasehantering till kontinuerlig leverans

Linus Aarnio

Supervisor : Manali Chakraborty
Examiner : Kristian Sandahl

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

When shortening release cycles and moving towards continuous delivery, a different approach for quality assurance may be needed than in traditional release management. To allow the transition, all stakeholders must retain a sense of confidence in the quality of release candidates. This thesis proposes a definition for confidence consisting of 30 confidence factors to take into account to ensure confidence from all stakeholders. Confidence factors have been found through interviews with 11 stakeholders, analyzed and categorized using grounded theory analysis. The found factors are grouped into two main categories: Process and Verification Results.

The thesis additionally contains a literature review of quality measurements and explores how confidence can be expressed in a continuous delivery pipeline. It is found that it is not possible to comprehensively express confidence only with metrics displayable in a pipeline when including only currently well-researched metrics, but with the combination of processes known to be followed in the organization some metrics provide coverage for many of the confidence factors.

Acknowledgments

This thesis would not exist without the help and support of many people. I want to thank Kristoffer Berglund and Kim Gunell at Crosskey for allowing me to work with them, coming up with the base idea, and being of great assistance in forming the first version of the idea into the plan it finally became. I also want to give a big thank you to both of them for all the motivational as well as practical support during the months this work was carried out.

I want to thank all the respondents from the interviews for taking time out of their regular work to participate in this study and for providing such valuable answers.

I also want to thank my supervisor Manali Chakraborty for finding examples and answers to all my method-related questions and for explaining that things do not always happen quickly in research, as well as my examiner Kristian Sandahl for motivational cheering at the small victories in the weekly letters and good feedback.

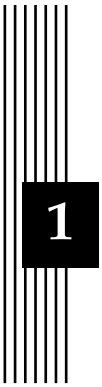
Finally, I want to thank my wife for telling me I was wrong when I thought I could not finish this work. Had you not told me that, I might not have been wrong.

Linus Aarnio
Linköping, August 2022

Contents

List of Figures

List of Tables



1 Introduction

The introduction aims to give the reader an understanding of how releases have traditionally been handled within large organizations, how they are proposed to be improved by modern practices, and the challenge of establishing confidence in a release candidate. The research questions and delimitations for this thesis are presented.

1.1 Motivation

The traditional way of delivering software is to plan a few releases each year, each release preceded by a staging period to make sure all internal dependencies are satisfied and perform manual testing. The releases are done with downtime, where all systems are taken offline and scripts are run manually to upgrade to the new version. New methods, such as continuous delivery allow releases to be more frequent and more automated. It is also possible to do deployments without downtime [11]. Continuous delivery involves setting up a pipeline that performs automated testing, building, and deployment of the software. Several prominent organizations have moved from the traditional model to different rapid release models [31]. To allow for this transition, the organization must ensure that a similar level of confidence is attained for each release candidate even though the release cycles are shorter. To ensure this, the organization must know which factors stakeholders take into account to establish confidence as well as which methods are available to satisfy those factors.

1.2 Aim

Crosskey is a company providing banking systems to Nordic banks. They use a traditional release cycle where five releases are planned each year, each release preceded by a month-long staging period where versions are fixed and extensive manual testing is performed. The goal of this thesis is to analyze how one department within Crosskey can ensure confidence levels are preserved if they were to move to shorter release cycles and ultimately continuous delivery. This improves the chances that there is an organizational acceptance of the new process and improves the chances for a positive outcome of the transition. Since several of Crosskey's customers are highly involved in the release process and continuously get information about quality assurance efforts, their acceptance of a new process is needed as well.

The thesis will investigate how confidence levels can be maintained in both external and internal stakeholders.

1.3 Research questions

1. Which factors do stakeholders in an organization take into account in order to establish confidence for a release candidate?
2. Which quality measurements are most effective in a continuous delivery pipeline to establish confidence in a release candidate?
3. How can the confidence level of a release candidate be expressed in a continuous delivery pipeline to stakeholders?

1.4 Delimitations

The study will be done with a focus on a specific department in Crosskey, the Capital Markets department. The starting point will be the process and product of this team. All definitions of confidence are based on this department and are not guaranteed to be applicable to other organizations. Due to time limits, the scope of quality measurements included to answer research question 2 and 3 only extends to those examined by papers included in existing literature reviews.

2 Theory

This chapter presents theory and central concepts related to continuous delivery and quality assurance of releases.

2.1 Continuous Delivery

Continuous delivery (CD) is the practice of developing software in short cycles, and utilizing automation to ensure that the newly developed software is ready to release [11, 48]. It is part of a group of practices called continuous practices which additionally includes continuous integration (CI) and continuous deployment.

2.1.1 Difference from other continuous practices

The continuous practices are highly related. Continuous integration focuses on frequently integrating newly developed source code to shared repositories, building and testing the code. Continuous delivery uses all practices of CI, but adds additional steps to enable the resulting build to be deployed to production, creating a release candidate but does not perform deployment. Continuous deployment adds the step of automatically doing the deployment for every release candidate [48].

2.1.2 Benefits

The benefits of adopting CD have been widely studied. One natural and important result of implementing CD is that it shortens the time until the customer can see the resulting product. This enables earlier opportunities for feedback which has been perceived as a major benefit by both developers and customers. Fewer errors are found in production, they are instead caught in the CD pipeline and are fixed before deployment to production. This results in a higher product quality for the customer. Productivity and efficiency improve when developers and testers spend less time setting up and maintaining manual test environments. They also improve since there is significantly less effort required to deploy software to production and there is a lower risk of being forced to pause all other work to troubleshoot issues that may arise in production after a release [11, 26].

As mentioned above, the risk of major release failure is reduced compared to traditional releases. This is the result of a reduced number of manual steps involved in the release, the reduced difference between environments resulting in fewer environment specific errors, increased testing of the deployment process itself and a decreased number of code changes in every release [11, 26].

2.1.3 Problems

While CD solves several problems for the organizations who adopt them, it also introduces new problems. In a systematic literature review by E. Laukanen et al.[30] 40 different problems were identified and classified into themes. The three main themes were, in order of decreasing prominence: testing, integration and system. One of the main issues found was ambiguous test results, which means that the test result does not guide to action with a clear pass or fail. Another was flaky tests, which means that the tests might pass or fail randomly and can not be trusted. In addition to this, testing problems include test suites that take too long to run to be practically usable and specific issues such as the need for hardware- or UI testing which introduces complexity.

Integration problems arise when code cannot be seamlessly merged into the mainline, causing merge conflicts and/or broken builds. These issues can arise when CD is introduced because of long-running tests causing developers to wait longer before committing to the mainline or flaky tests preventing the build from passing and in turn making the time until merge to mainline longer.

System design problems means that the system might not be directly compatible with processes needed for CD. It was found that if a system is not designed for the processes needed for CD issues such as complex builds and lack of testability arise.

2.2 Quality measurements

This section describes and explains the theory behind different quality measurements which are researched as part of the literature study described in section 4.3.

2.2.1 Software metrics

There exists a range of software metrics used for fault prediction which in a systematic literature review performed in 2013 by Radjenovic´ et. al. [41] was grouped into the following categories:

1. **Traditional:** size and complexity metrics.
2. **Object-oriented:** coupling, cohesion and inheritance source code metrics used at a class -level.
3. **Process:** process, code delta, code churn, history and developer metrics.

Of these, the one found to be most commonly used in the reviewed papers was object-oriented metrics, followed by traditional metrics and the least used was process metrics.

2.2.1.1 Traditional metrics

Size metrics is dominated by measuring *lines of code* (LOC). It is trivial to measure. Complexity metrics can include size, but also factors such as the control structure of the software where a higher number of decision paths indicate a higher complexity. A higher complexity makes comprehension of the software more difficult which in turn makes the development and testing more difficult. A common measure of complexity is McCabe´s cyclomatic complexity.

McCabe´s cyclomatic complexity is based on a graph representation of the control flow of the program. The graph is built by letting nodes be blocks of code in the program and edges be branches in the control flow. The cyclomatic complexity v is defined as

$$v = e - n + 2p$$

where e is the number of edges, n is the number of nodes and p is the number of connected components (units)[33]. When measuring a single unit (program, method, etc), p is equal to 1 which is a common situation. It can be hard to distinguish complexity metrics from size metrics, and some consider complexity to be a form of size metric as it is highly correlated to LOC [35, 18].

2.2.1.2 Object-oriented metrics

Object-oriented metrics are the most researched form of software metrics, prevalent in approximately two times as many papers than traditional and process metrics in the literature review by Radjenovic´ et. al. [41]. There exists a large range of object-oriented metrics, with the most common being the *CK metrics suite* proposed by S.R. Chidamber and C.F. Kemerer in 1994 [13]. This includes several metrics which are described in Table 2.1.

Metric	Abbreviation	Description
Weighted Methods Per Class	WMC	Measured by calculating the complexity of each method in a class using some complexity metric and adding the measurements together to form a WMC metric for that class.
Coupling Between Object Classes	CBO	Measured by counting the number of collaborations for each class
Response For Class	RFC	The number of different methods that can be executed when an object of the class receives a message
Lack Of Cohesion In Methods	LCOM	Measures the cohesion of methods through connected components. A connected component is a set of releated methods, where two methods are related if they access the same class-level variable or call each other. A higher score indicates lower cohesion.
Depth of Inheritance Tree	DIT	Measured for a class by following the inheritance tree of the class hierarchy and recording the greatest depth.
Number Of Children	NOC	Measured for a class by counting the number of children of the class.

Table 2.1: CK metrics as described in [13]

2.2.1.3 Process metrics

Process metrics are usually based on either traditional or object-oriented metrics, but instead of measuring the software only at a given point of time, they give information about the change in metrics value. Process metric can be either of the kind *code delta* or *code churn*.

Code delta is the difference between two builds as computed for a particular metric [21]. If we apply code delta to the simplest metric, LOC, then the measurement would represent the number of lines of code added or removed between the builds. A weakness with code delta is that it fails to express changes of a kind where 5 lines are removed and five lines are

added which would represent a code delta of 0. To solve this, code delta can be supplemented with code churn.

Code churn adds the metric value of additions to the absolute metric value of deletions. For our simple example, this would mean a code churn LOC value of $|5| + |-5| = 10$. Similarly to code delta, code churn can be used for any metric. Together, code delta and -churn represent the evolution of a system in regard to how much it has changed since a given point [21].

Process metrics can also measure other data about the history of the source code. These include number of past faults, number of changes, the age of the changed module and the number of modules changed together with the module (change set). Additionally, they can involve developer metrics such as the number of developers who changed the file or the number of new developers who changed the file [41].

2.2.2 Code smells

Related to software metrics is the concept of code smells. Both categories use quantifiable properties of the source code to perform fault prediction. Code smells are signs of bad software development practises that can be seen in the source code. The term was coined by Martin Fowler and Kent Beck, possibly under the influence of the odors of Beck's newborn daughter [19]. The occurrence of a code smell indicates that there may be deficiencies in the design and/or programming style of a software system. Multiple tools exist to discover code smells using different methods, but the most common and most fitting for automation is metric-based. The different code smells are described in Table 2.2.

Table 2.2: Code smells as described in [19]

Code Smell	Symptom	Reason for being a smell
Duplicated Code	The same code structure exists in more than one place.	Bugs can be introduced if the code needs to be changed and the developer misses to change one or more of the places.
Long Method	There exists a method which is longer than some given limit on LOC.	The method is harder to reason about.
Large Class	A class is doing too much and has too many instance variables or too much code.	High probability of duplication.
Long Parameter List	A method is declared with many parameters.	Long parameter lists are hard to understand, they become inconsistent and difficult to use, they need change more often.
Divergent Change	One class is commonly changed in different ways for different reasons.	It becomes unclear which part of the system a change should be applied to.
Shotgun Surgery	Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.	It is easy to miss an important change.
Feature Envy	A method that seems more interested in a class other than the one it actually is in. An example is using many getters in another object.	The method is likely in the wrong class, and might be missed when the other class changes.
Data Clumps	Some data items are "clumped" together in many places.	Grouping data clumps together into classes reduces parameter lists and can open possibility to reduce Feature Envy.

Primitive Obsession	Primitive types are overused compared to record types.	Type safety and guaranteed validation opportunities are lost.
Switch Statements	There is an abundance of switch statements.	The same switch statement is often duplicated.
Parallel Inheritance Hierarchies	Every time you make a subclass of one class, you have to make a subclass of another. Recognizable through prefixes of class names in one hierarchy being the same as the prefixes in another.	Same issues as Shotgun surgery.
Lazy Class	Classes which do not do much exist.	The useless classes still need to be maintained.
Speculative Generality	The ability to do many things in the future has been built in, even though it is not needed. Recognizable by abundance of abstract classes, unused parameters and unused methods.	Increases difficulty to understand and maintain the code.
Temporary Field	Objects with instance variables which are only set in certain circumstances	The code is difficult to understand.
Message Chains	The client asks one object for another object, only to ask that for another object and so on.	The code gets tightly coupled to the structure of the navigation.
Middle Man	A class uses delegation in all or most of its methods.	The class becomes hard to reason about.
Inappropriate Intimacy	Two classes are coupled too tightly together, using many methods of each other.	The same problems as Feature Envy.
Alternative Classes with Different Interfaces	Classes with methods doing the same thing but with different signatures exist.	The code becomes harder to understand.
Incomplete Library Class	A library is used but does not cover the needs of the software.	The library might be exchanged for a developers own implementation or duct-taped.
Data Class	There are many classes with only fields, getters and setters.	They are probably manipulated too much by other classes, and that behavior should instead live in the data class to increase code quality.
Refused Bequest	A subclass is not using data and methods of its parents.	The code becomes confusing.
Comments	There is an abundance of comments in the code.	There is a high probability that the comments are there because the code is bad.

2.2.3 Testing

The basic way of preventing faults is software testing. This can be done either manually by using the software and checking if the functionality works as intended, or in various automated ways. The focus of testing may also differ, with goals such as verification of functionality, security, performance or other factors.

2.2.3.1 Automated or manual

Testing can be done in either automated or manual fashions. Automated testing includes unit-, integration- and system testing using specific code libraries such as JUnit [28] or REST-assured [44], as well as acceptance testing using scripted tests using tools such as Selenium [47] or Cypress [17]. Manual testing focuses on system- and acceptance testing and is done by using the software in specific ways to exercise test cases documented in a tool like REQ-test [43].

2.2.3.2 Unit testing

Unit testing is a common method of functional testing, often seen as the first step in a series of tests. It is usually performed by the developer writing the code. Unit testing starts with identifying a discrete unit in the software. This unit may be a function, a class or even a set of these classified together as a *Unit of work*, defined by Roy Osherove as "the sum of actions that take place between the invocation of a public method in the system and a single noticeable end result by a test of that system" [38]. The benefits of unit tests include that they are easy to understand and analyze, as well as to implement. This allows for early implementation in the development phase and frequent runs to be able to discover faults in the software at an early stage. However, for the unit test to have these characteristics the code needs to be written with testability in mind. It is important that the units are sufficiently isolated and that the expectations on their API are clear to allow for good unit tests [38].

2.2.3.3 Integration testing

In integration testing, the units are combined and tested to evaluate the interaction among them [25]. This is seen as a natural step after unit testing, and before system testing. Integration testing can use the same tools as unit testing.

2.2.3.4 System testing

System testing involves testing all of the software involved in the system together. It may test for functionality, but in addition to functionality this is often the stage where testing of factors such as security and performance can be performed.

2.2.3.5 User acceptance testing

User acceptance testing is the last stage of testing. The system is used in a realistic manner either by the end user or other stakeholder who has set the requirements. The main goal is not to examine specific details of the software as in earlier stage of testing, but to see if the software can deliver the expected business benefits when operated by its designated users [23]. The shape of this testing can vary, but to be considered a performed acceptance test there needs to exist an acceptance criteria and a structured way to perform the test to provide evidence of the acceptability of the system.

User acceptance testing is traditionally performed manually, but can be fully or partially exchanged for automated tests using tools such as Selenium or Cypress.

2.2.4 Test quality

There can be quality differences in tests, with two test suites of the same codebase being able to detect a differing number of faults. In the same way quality of code can differ, so can the quality of tests.

```
// Original
if (i > 10) {
    ...
}
```

```
// Mutant
if (i >= 10) {
    ...
}
```

Table 2.3: Example of a mutation where an arithmetic operator has been exchanged

```
// Original
for (int i = 0; i < 10; i++) {
    ...
}
```

```
// Mutant
for (int i = 0; i != 10; i++) {
    ...
}
```

Table 2.4: Example of an equivalent mutant

2.2.4.1 Test coverage

A common way of measuring test suites is *coverage* which commonly takes the form of statement- or decision coverage. Statement coverage is the measure of statements being run by the test suite, and decision coverage is the proportion of decision branches being run by the test suite. They can easily be measured by automated tools and provide a good overview of the extent of testing on the code base.

2.2.4.2 Mutation testing

There are other ways of measuring test suites, such as that of mutation testing. Mutation testing is a technique where small changes are introduced to the source code, producing *mutants*. These mutants have altered behavior with regards to the original source code. The test suite is run against the mutants, to measure a mutation score which expresses the degree to which a test suite can detect changes in behavior. The change introduced to the source code is called a mutation operator. A typical example of mutants is exchanging operators, such as the one seen in Table 2.3. There are several additional mutation operators [27].

There are inherent problems with mutation testing which have hindered wide industry adoption. The main problem is that the creation of large amounts of mutants requires large amounts of resources and time which makes it impractical to run often. Another core problem of mutation testing is that of equivalent mutants, where a mutant created might have equivalent behavior to the original source code and thus should not be detected as a failure by the test. An example of a created equivalent mutant can be seen in Table 2.4. There are techniques available to mitigate these problem which allows mutation testing to be automated and scalable [27].

2.2.5 Code review

Code review is the process of not merging commits to production branches before it has been manually inspected and approved by one or more developers other than the author. What this inspection includes varies between practitioners, both between organizations and within organizations. The process of code review is mainly used to ensure quality and maintainability of the code itself and to share knowledge between developers, but it is also seen as a way to detect faults in the product [7].

Code review before merging can be optional or mandatory within an organization. Mandatory code reviews can be enforced by popular version control management software such as Gitlab, Bitbucket and GitHub [15, 20, 4].

3 Related Work

3.1 Related work

This chapter describes other work on software quality and confidence in continuous delivery, and compares it to the work of this thesis.

3.2 Quality assurance in continuous delivery

The quality assurance practices to be followed by a QA professional in continuous delivery have been investigated by Cheriyan et. al in 2018 [12]. Based on a literature review, they propose a model called ACID-QA, with the six steps: Fitness Function, Continuous Delivery Readiness, Quality Assurance in the Pipeline, Quantitative and Causal Analysis, Learning Development and Team Culture. The step most related to this thesis is Quality Assurance in the Pipeline. They propose several practises, tools and frameworks to implement in the continuous delivery pipeline. These include practises such as unit testing and threat modeling, tools such as Checkstyle for static code quality and frameworks such as the testing pyramid for determining proportions between different test activities. However, they provide no reasoning for why these specific practises, tools and frameworks are chosen and what benefit they provide to the quality assurance process but instead they are simply listed. There is also no description of the method and extent of the literature review, which unfortunately lowers the credibility of the study. This work aims to build upon the quality process with clear motivations on why to include quality practises and scientific backing of the effect from chosen practises and metrics.

The quality of a candidate is expected to mostly be measured by different test activities. D. Ståhl et al. have written a paper presenting a model for including test activities in CI and CD, called the Test Activity Stakeholder (TAS) model [32]. The model suggests different test activities to support the stakeholder interests *Check changes*, *Secure stability*, *Measure progress* and *Verify compliance*. The importance of these interests to be validated was found through interviews with participants from four different companies. For each stakeholder interest the authors use another round of interviews to determine whether the interest is best supported by unit tests or system tests, whether it is best supported by tests run in a simulated test environment or on real hardware and finally whether it is best supported by automated testing

or manual testing. In total, 25 interviews were conducted. The result is the TAS model which presents a set of test activities to be performed in different stages of the pipeline to properly support all stakeholder interests. The paper is delimited by only including large companies with more than 2000 employees, which all develop complex products that also "include a significant amount of mechanical and electronic systems". The test activities suitable for this kind of company may differ from the kind of test activities suitable for a company like Crosskey which has less than 500 employees and purely software products. The TAS model presents which kind of tests should be used for the different stakeholder interests, but does not present how to ensure the extent of testing is enough to guarantee that the stakeholders can have confidence in the release candidate.

3.3 Confidence

A major trade-off that has to be considered is that between velocity and confidence, and deciding how high the level of confidence has to be to be enough. This is explored in a paper by G. Schermann et al. [46]. They define confidence as "the amount of confidence gained on the three quality gates automated testing, manual testing, and code reviews" and describe the need to include multiple factors when quantifying confidence, but do not attempt to formulate how to do this quantification. They define velocity as "the pace with which changes are running through the quality gates, starting with the commit of a change until it reaches the production environment" but simplify it to "the time needed to assess each single quality gate and to build and deploy the application" including time for manual decisions for deployment and maintaining automated test suites. They present four categories a company can belong to with regards to these metrics: *Cautious* where the company maintains high confidence but low velocity. The case studies describe having 100% test coverage and six to eight weeks of time reserved for staging period. This was shown to lead to an unmaintainable test suite which ultimately decreased release confidence. A company in the *Problematic* category has both low velocity and low confidence. A company in the *Madness* category has high velocity, but no confidence due to lack of quality assurance. A company in the *Balanced* category has both high velocity and high confidence in releases. It is described as the vision for continuous delivery, and ways to transition towards it from both Madness and Cautious are presented. This paper describes the importance of knowing the confidence of a release, but does not present how it should be measured and quantified which would be an improvement for a company uncertain of the category it falls into. The authors propose this kind of quantification as suggested future work.

The information needs in continuous integration and delivery have been explored in a paper by Ahmad et al. [2]. The authors performed a multiple case study to identify the type of information that different stakeholders seek, and found confidence to be one of the main categories of information. There were three specific information needs in the confidence category:

1. How much confidence do we have in the release to deploy to the customers?
2. How much confidence do we have in the test suites?
3. How much confidence do we have in stand-alone projects to be merged into the master branch/baseline?

Which together were of interest to all stakeholders (development, testing, project management, release team, compliance authority). The confidence questions were among the information needs considered most important and most frequently mentioned, and at the same time requiring the most effort and time to answer compared to other identified information needs. The study regards confidence as an informed opinion by the stakeholders

based on an aggregate of all data sources. Additionally, the authors discuss that there is a need for an understanding of the concept of confidence, including which information is needed and from which sources as well as to which extent it can be calculated through automation. They note that these needs are "notoriously unquantifiable". This thesis aims to add to the work by answering the question of which information is needed, as well as beginning to explore which data sources could satisfy these needs. Good knowledge of the specific factors to consider may reduce the time and effort needed to find information in this category.

The conclusion from the related work is that confidence is considered important in continuous contexts, but is not properly defined and therefore hard to measure and guarantee for all stakeholders.

4 Method

This chapter describes the methods used for reaching an answer to the research questions. It describes the research design as well as details on how the data gathering and analysis was performed with the intention to promote replicability of the study. Additionally, it motivates the choices of each research method to assure the conclusions are trustworthy.

4.1 Research Design

The study consisted of three research activities, which each answered one of the research questions. Two of the activities were performed separately, and the third consisted of combining results from the others. The relationship between research methods and the research questions can be seen in Figure 4.1. The method to answer RQ 1 is described in section 4.2 and results can be found in section 5.1, the method to answer RQ 2 is described in section 4.3 and results can be found in section 5.2. The resulting combination of these to suggest data sources for confidence factors can be found in section 5.3 with Table 5.10 displaying the mapping between them.

4.2 Grounded theory approach to confidence factors

To answer RQ 1: *Which factors do stakeholders in an organization take into account in order to establish confidence for a release candidate?* a qualitative approach using interviews with stakeholders and Grounded theory analysis was used.

4.2.1 Interviews

The main data gathered for the study was interviews with stakeholders. This section describes the interview process.

4.2.1.1 Subject selection

The interview subjects were chosen based on roles selected to represent different groups that had to be confident in a release candidate for it to be considered ready for deployment. The base selection was

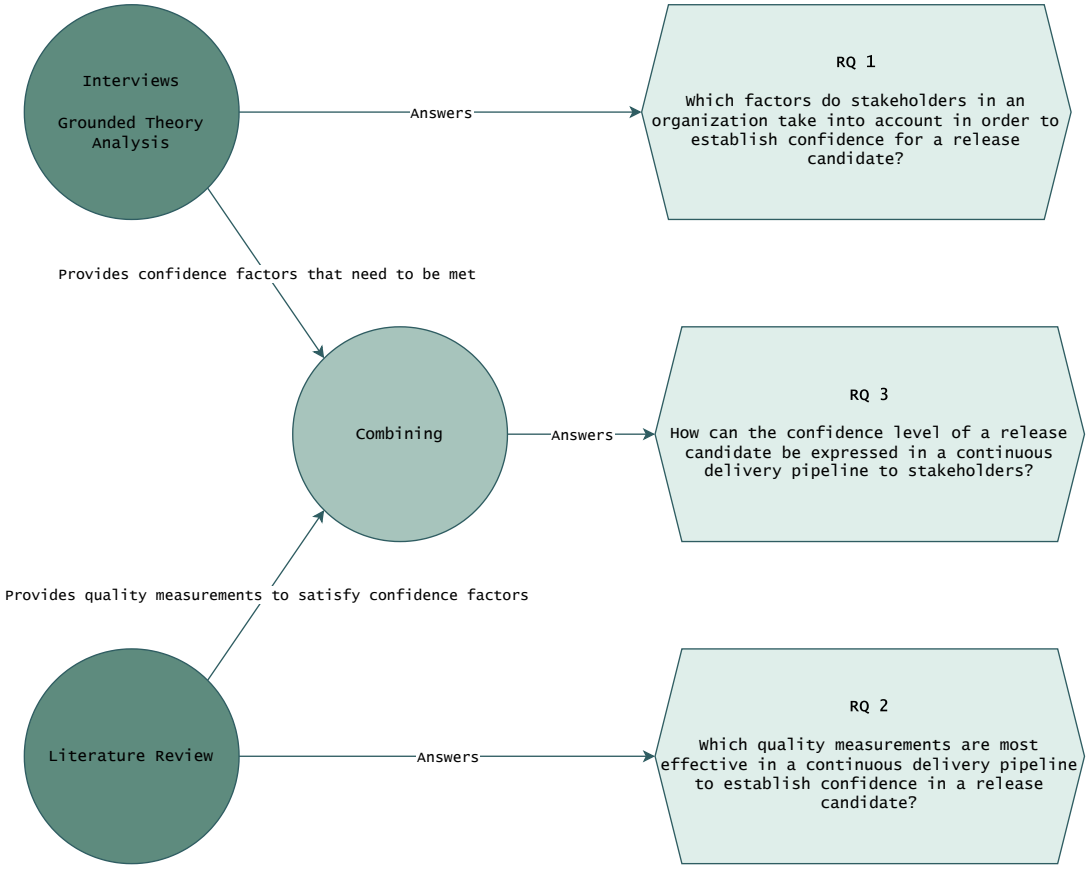


Figure 4.1: The research methods used in the study and their relationship to each other and to the research questions

- developers
- managers
- customers
- product specialists

With the main focus being on product specialists as the role has the main responsibility for quality assurance in the release process today. While selecting candidates for the roles, the selection was widened to include specialized roles which did not fit into this enumeration such as the chief security officer of the company. The interview subjects were found through a snowballing process, where the researcher initially asked for recommendations from the company supervisors and then for more recommendations in the end of each interview. This allowed the search for new candidates to continue until no new recommendations could be gathered. All of the asked candidates agreed to be interviewed. A total number of 11 interviews were performed, where some candidates left the perspective of multiple roles: one product specialist/ previous developer and one manager/ previous customer. In addition to this five product specialists, one manager, one chief information security officer and one customer was interviewed.

4.2.1.2 Interview structure

The interviews were semi-structured, a choice motivated by the lack of known factors of confidence. As stated by W. Adams [1], one of the situations where semi-structured interviews

(SSI) should be especially considered is when "examining uncharted territory with unknown but potential momentous issues and your interviewers need maximum latitude to spot useful leads and pursue them" which is an accurate description of the situation of this study where a lot of confidence building can be assumed to have been implicitly embedded in the old release process and could potentially be missed without the possibility to probe into details of the subjects thoughts.

An interview guide was created to provide a base structure for the interviews. As suggested by Adams, the interview guide was not design to be read verbatim, but instead to act as a starting point for the probing. It consisted of four main questions and nine suggested follow-up questions to the question considered by the researcher to be most important. The interview guide was built as:

- Can you describe what it means for you to have confidence in a release candidate?
- What do you do to establish confidence in the quality of a release candidate today?
  - Which manual measures do you use?
    - How much do you trust the result?
      - Why do you/ do you not trust it?
  - Which automated measures do you use?
    - How much do you trust the result?
      - Why do you/ do you not trust it?
  - What do you look at in the software itself?
  - What do you look at in addition to the software? (E.g. environments, number of changes, etc. etc.)
  - Are all these metrics general (same for all software products), or product dependent?
- Describe a time when the quality assurance of a release has been lacking.
- What would make you decide that a release candidate is not fit for deployment?

In addition to the follow-up questions suggested in the interview guide, the researcher was prepared to ask additional follow-up questions depending on the themes lifted by subjects in order to explore as wide areas as possible. In accordance with the method described by Adams as well as the theoretical sampling methodology described later in this chapter which is a part of grounded theory [16], the interview guide was re-evaluated after initial analysis of each interview. This resulted in a few alterations and additions, seen together with the motivation for introduction in Table 4.5.

4.2.1.3 Interview implementation

The interviews were carried out in person for the respondents who worked at Crosskey headquarters in Mariehamn. For respondents who worked in another office or remotely, the interview was carried out over Microsoft Teams. Each interview lasted between 30 and 60 minutes, depending on how much the respondent had to share and the amount of discussions that arose in addition to the questions in the interview guide. While answering the main questions, each topic mentioned which had importance for the respondent's confidence led to a follow up question to extract details.

Before each interview, the respondent was briefed on the conditions. They were informed about the intended use of the information gathered from the interview and about the fact that their responses would be anonymous. They were also asked for consent for recording

Original (or '-' if new addition)	New	Motivation
Can you describe what it means for you to have confidence in a release candidate?	If you tell someone you have confidence in a release candidate, what does that mean?	Respondents did not understand the question as it seemed too broad and abstract. The updated variant allowed them to focus on their subjective view of confidence which was the desired result.
In context of manual measures: How much do you trust the results?	If the manual test suite has passed, how confident are you that everything will work after the release?	It was discovered that respondents would say they trusted the results of individual manual tests, but later added factors such as new additions to the code base which still made them unsure that the software was fault-free.
-	If the manual regression tests would be fully replaced with automated tests for equivalent test cases, how would that change your confidence and with which conditions?	Many respondents had little or no experience with automated-Sttesting which made the question about their existing confidence in test automation provide poor answers. This speculative question allowed them to share their views on trust in automated tests.
-	As a final question: Is there anything else concerning the confidence in a release candidate you would like to share?	After being interrupted by a subject when attempting to wrap up the interview, the researcher noted that there might be factors the subjects have thought of and waited for the right time to share
-	What impact does the proximity of a change to the release date have on your confidence in the quality of the release candidate?	It was found that different factors relating to proximity to the release date affected the confidence in release candidates, asking for this allowed for more findings in the area
-	Do you always have confidence that you know what will be included in a new release?	During the early responses to the question of when a release had been lacking in quality assurance, a reason that came up was that a change introduced in the release was not known to the person performing QA.

Table 4.5: Updates and additions to the interview guide

of the interview for transcription and analysis purposes. All respondents gave consent to recording which was performed with a mobile device for the in person interviews and using the built in recording function of Microsoft Teams for the remote interviews. The recordings were transcribed manually by the researcher.

4.2.2 Analysis

The data was analyzed using Straussian Grounded Theory (GT) as described in Basics of Qualitative Research [16]. This method was chosen because of the subjective nature of the question about how confidence is established for a release candidate. As stated by Corbin and Strauss "The procedures can be used to uncover the beliefs and meanings that underlie action, to examine rational as well as nonrational aspects of behavior, and to demonstrate how logic and emotion combine to influence how persons respond to events or handle problems through action and interaction". The purpose of the method is to construct theory from qualitative data.

4.2.2.1 Open coding

The first step of analysis consisted of open coding. It is the process of going through the data and labeling pieces of data with a code representing the meaning of that data. For this thesis, the open coding stage was performed on paragraphs in the interview transcripts. Each interview transcript was imported to the software NVivo [37]. The paragraphs were then added as references to nodes, where each node represented a concept. For each paragraph, a new node was created if the concept could not fit under an existing node, otherwise a reference was added to the existing node which covered the concept.

A primary open coding round was performed on each transcript after the interview, following the best practice in GT of immediate and continuous data analysis. This allowed the researcher to find opportunities for theoretical sampling, discussed later.

4.2.2.2 Axial coding

After open coding another coding activity was performed, called Axial coding. In axial coding, the researcher attempts to identify relationships between the existing categories. This is the step which takes the analysis from describing what is happening, to offering explanations and theory. In this thesis, it was done through a second round of analysis of all references in each node. To find relevant context, an analytical tool from Corbin and Strauss called *the paradigm* was used. The paradigm consists of looking for *conditions*, *actions-interactions* and *consequences or outcomes*.

- **Conditions** answer questions of *why*, *when* and *how come*. In interviews the respondents usually talk not only about their actions, but also about the reasons for these actions. Key words suggested by Strauss for identifying conditions are *because*, *since*, *due to* and *when*.
- **Action-interactions** are the actual respones to what a person is doing in a certain situation. They are here analyzed in the context of which conditions they are performed in, and which consequenses they have.
- **Consequences** are the outcomes of action-interactions. Strauss highlights that these are not only the actual outcomes, but the *anticipated* outcomes as well. An anticipated outcome can be seen as the reason the respondent is doing something, which might not always be the actual result. Sometimes, the difference between the anticipated and the actual outcome is an important piece of the context as well.

The nodes sharing similar conditions and/or consequences were then grouped together to form more over-arching categories.

4.2.2.3 Memoing

Memoing is an important part of GT where the researcher writes memos to capture parts of the theory as it emerges. During the process of this study, memos where written following

the analysis of each interview to describe the researchers immediate impression of the main theme in the interview. Additionally, memos were written during axial coding to describe the intuition for the context analysis and the formed group. The memos are what forms the basis of the result part of this thesis together with the raw data.

4.2.2.4 Theoretical sampling

Theoretical sampling is a GT concept which describes the process of gathering data based on gaps in the emerging theory identified during analysis, and gathering data until the concept can be seen as saturated. The theoretical sampling can be both data of the same kind, as well as different kinds. In this thesis, theoretical sampling was performed in several ways, described below.

- Identifying factors not mentioned before during the initial analysis and then asking direct questions about the same context in subsequent interviews. These can be seen in Table 4.5.
- Asking for, and collecting documentation of current and planned practices. This included the documentation of test cases exported from the tool REQ-test [43] as well as a powerpoint presentation describing a new suggestion for security practises.
- Asking for recommendations of other stakeholders to interview, with regards to the points that had been taken up in the interview. This included both other respondents who could give different viewpoints on the same factors as the asked respondent, but also respondents who could give more information about concepts mentioned by the asked respondent as important but which they did not know enough about to elaborate on.

4.3 Literature Study of quality measurements

To answer RQ. 2: *Which quality measurements are most effective in a continuous delivery pipeline to establish confidence in a release candidate?*, a semi-structured literature study was performed on the subject of software quality measurements. The choice to not perform a full structured literature review was based on a tertiary review of software quality research performed in 2021 by Champion et. al. [9] which found many existing secondary analyses of high quality. Therefore, this work was done by going through existing literature reviews and making a selection for the quality measurement methods that are possible to include in a continuous delivery pipeline and have been consistently found to be effective for fault detection.

4.3.1 Source material gathering

The literature was found through two main sources: direct search and snowballing. For the quality measurement areas where the material found in the searched literature reviews was not deemed sufficient, additional sources were gathered through search. The snowballing was performed starting with gathering secondary sources cited in [9] and then continued with gathering primary sources cited in those secondary sources. The direct search was performed through the Linköping University library using EBSCOhost to search through a number of databases, including but not limited to IEEEXplore digital library, ACM Full text collection and arXiv. The search terms used are found in Table 4.6.

4.3.2 Selection

A selection was made on quality measurement methods to include in the study. The criterion was that the method had to:

Search terms	
unit testing	manual testing
automated manual testing	software fault prediction metrics
software fault prediction metrics	continuous delivery quality assurance
code review	mutation testing
test environments	test environment quality

Table 4.6: Search terms used for additional literature review material gathering

1. Be usable in a continuous delivery pipeline
2. Have been consistently found to perform well in the context of fault prediction

From the sources found through the source material gathering, a loose selection was done based on excluding sources where the title or abstract made it obvious that no methods fulfilling the criterion was included in the paper. For the studies not included in an existing structured literature review, a quality assessment was performed to verify that the study had low bias and high internal and external validity.

4.4 Mapping of quality measurements to confidence factors

To answer RQ. 3: *How can the confidence level of a release candidate be expressed in a continuous delivery pipeline to stakeholders?* the analysis of confidence factors was continued together with the knowledge about quality measurements to map which confidence factors could be expressed by different quality measurements. No new data was introduced to answer this question, it relied exclusively on the results from the rest of the study as seen in Figure 4.1.

5 Results

This chapter presents the findings that were produced by following the method described in chapter 4. The results are only presented in this chapter, and are analyzed and discussed in chapter 6.

5.1 Grounded theory approach to confidence factors

Through the grounded theory analysis, a total of 30 codes were created in the open coding of the interviews with each code representing a confidence factor. The number of references for each code can be seen in Table 5.7. These codes were then split up into 11 categories during axial coding. Finally, the categories were connected into 2 main categories. This section will present the found categories in a top-down fashion, with description of the data collected in interviews for each category and explanation of the connections between them.

5.1.1 Process

The first main category for confidence is Process. The category structure can be seen in Figure 5.1. The category contains all categories which relates to *how* things are done, and how they change. The analysis has found that a large part of the confidence for a release candidate is built through process factors, and it also impacts how much confidence is placed in the results from the second main category, Verification Results. The rest of this section will describe the subcategories of Process.

5.1.1.1 Differing needs for QA depending on the change

This category results from the findings that multiple stakeholders place importance in what, how much and how the software has changed from the last release. This is clearly seen by the example where one of the respondents found it hard to pinpoint what confidence meant to them, and after spending some time without coming up with anything concrete they said "If absolutely nothing has changed, I will have confidence in the release candidate!". Although this statement was said in a comic fashion, the point is undeniable. As will be seen, and as the same respondent confirmed, there is a gradual scale after the no-change case where confidence decreases with increasing change.

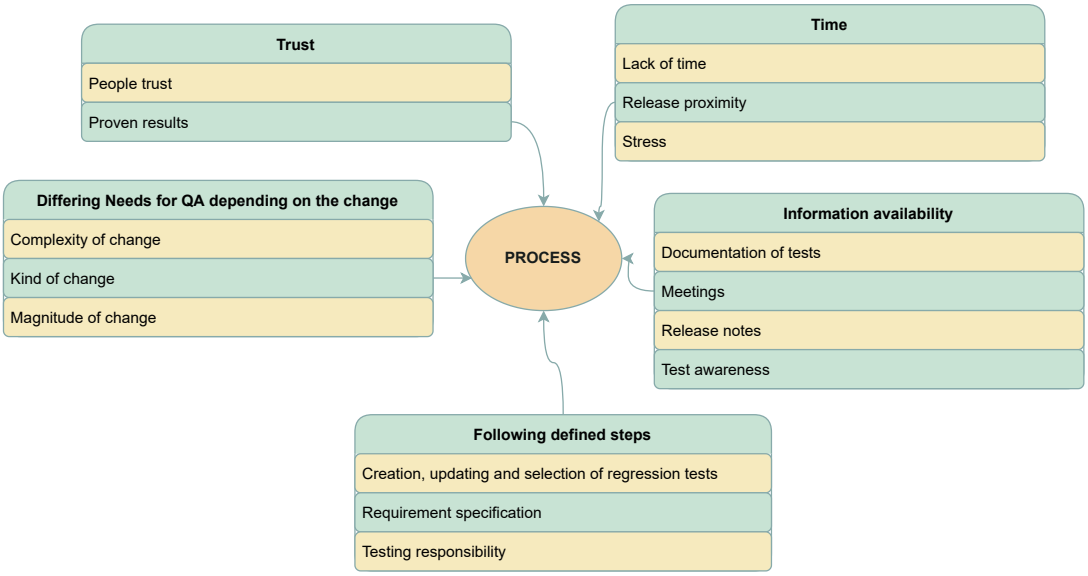


Figure 5.1: The categories under the main category Process

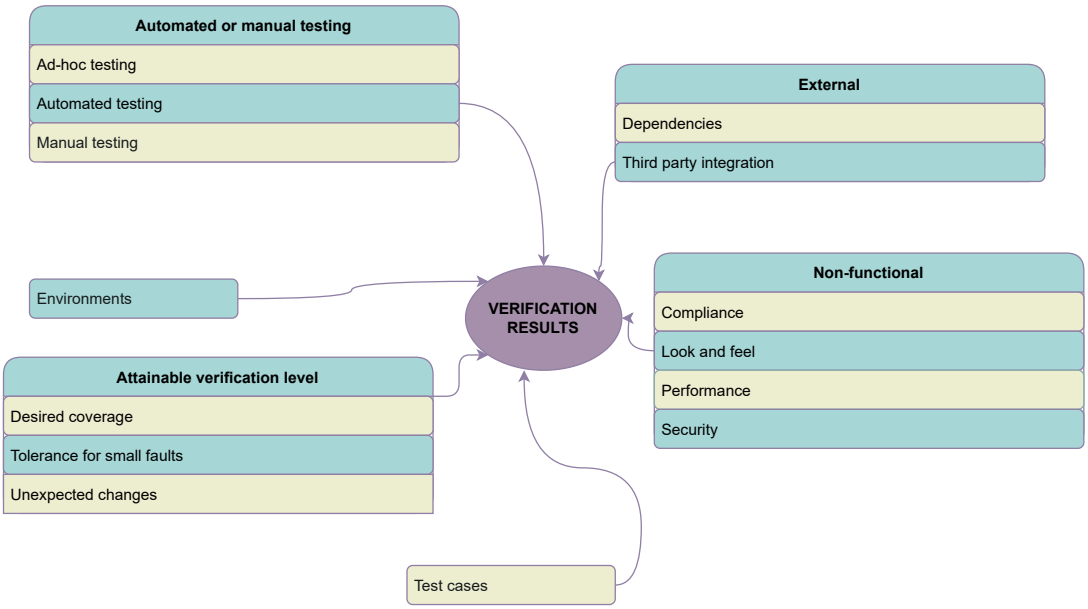


Figure 5.2: The categories under the main category Verification Results

Magnitude of change

The first change factor is the magnitude of change. A release with a higher level here can have both a larger number of individual changes, or more code changed with each change. Stakeholders from all categories have named this as a factor for their confidence. There is a consensus that "the more changes you have, the higher the risk is for the release to fail". The reasons for some are simply that there is a statistically higher risk that a fault might be present when there is more code deployed, but the increased need for quality assurance is also mentioned: "If there has been a big change in functionality you will have less overview of what has happened and less opportunity to carefully think about how existing test cases

should change regarding to the changes and what is affected. That decreases confidence".

Kind of change
In addition to the magnitude of change, the kind of change also matters for stakeholders. This includes:

- Whether the change is a bug fix or a new feature
- Whether the change is cosmetic or functional
- How big proportion of the users that are affected by the change
- Which system is affected by the change

The reasons for this varies. One customer representative said they have less confidence in new functionality than bug changes, because of the differing difficulties of communicating expectations on the desired results. They said that "For new functionality, something that is obvious to the bank might have big holes if the developer has not understood it". There is also more uncertainty about what should be included in testing for new functionality whereas for a bug fix the sentiment is that regression testing combined with verifying that the bug is fixed provides enough confidence. This adds to the fact that new features have a more negative impact on the confidence than bug fixes. Stakeholders also agree that there is a considerably greater need for testing for a cosmetic change than a functional one, and in the same way there is a greater need for testing of a change that affects a large proportion of the users as opposed to a limited subset of users. Which system is affected by the change is a question that ties in to other categories found in the study. Stakeholders have mentioned that they have less confidence in a change introduced in systems with a previous history of faults, one that the developer in question has not worked on before or one that is unusually complex or critical for the functionality of the product.

Complexity of change
When a complex change is included in the release, more testing is needed for the stakeholders to feel confidence. One customer representative said it was hard to determine how complex a change is, system-wise. This made the respondent unsure of how much testing was needed, and lowered overall confidence in the release when the decision was made to treat the change as not-complex while not knowing the actual complexity. The respondent wished for developers or other roles with equivalent insight to place risk scores on changes, to determine how much quality assurance was needed.

5.1.1.2 Following defined steps

The confidence in a release candidate increases when a set of defined steps are followed as part of a QA process. Stakeholders find that several steps are crucial to include in a process, these are detailed in this subsection.

Requirement specification
Respondents found it important that requirements and expectations on the software are clearly specified before the rest of the QA process. A developer said that they always wish to have requirements specified to the level of "here is how the response should look like in this api". This allows them to be confident that the developed result does what is expected and is also a prerequisite to have confidence in automated tests created by the developer. The same respondent said that a detailed specification allows them to "build in a way where what we produce does very little which makes it easy to look at e.g. a service, say 'what is this supposed to do?' and test the exact specifications". In the same way, a lack of detailed

requirements was described as leading to situations where tests would pass but the product might still be faulty.

This view is shared by other stakeholder categories. Product specialists reported an expectation of less faults in the production environment when they were allowed to spend sufficient time on requirement specification and test case design before any development had started. When development started before all requirements had been specified, it lowered confidence throughout the chain all the way to release. Test cases which were specified during the testing phase as opposed to during the planning phase were reported to be less trustworthy. Additionally, having the requirements and test cases discussed by multiple people before development started also increased the confidence in the result.

Testing responsibility
Test may come from several directions. A system can be said to be tested if the developer has created automated tests based on their understanding of the system. It may also be tested based on a detailed specification from the product specialist, either performed by the product specialist themselves or implemented by a developer. Respondents agree that there is a higher confidence placed in the results of testing when it is planned by the product specialist who has created the requirements for the feature. However, they did not find it important in which way and by whom the test cases were implemented as long as it was clear which specific test cases from the plan were performed. Similarly, a developer said that they placed high confidence in a release if both a product specialist and a customer had marked it as verified: "If they say it is okay, then it is okay".

Creation, updating and selection of regression tests
For the results of a test suite to be trusted, there is a need to know that test cases have been created for new functionality or updated for changed functionality. Product specialists in Crosskey rely on a regression suite consisting of both manual and automated tests which are performed before every release. When new functionality is included, respondents have said that it is important to include new tests for this, which is up to each product area to do. The update of test cases is perceived to provide the most confidence when done in the planning phase as opposed to during the testing phase.

When it comes to selection, there is a prioritization where some test cases are seen as essential for every release to be confident in the quality, while the rest are chosen "if one feels it is needed". In the same way, a product specialist is confident that they have tested enough when they have thought through what functionality might change because of a software change and tested that. One respondent said that "nothing should be untested of course, but I think it is important to not just test individual scenarios but to test the entire environment. To have thought through how the users daily jobs look like and see if anything might have changed there".

5.1.1.3 Information availability

Information availability is the extent to which a stakeholder feel they know enough about what has changed in the release, what has been tested and what the results are. It builds upon the other factors and can make or break them; a release which does not have any faults and has been thoroughly verified might still be given a low confidence grade by a person who feels they do not have enough information about it. The findings in analysis about information availability consists of four subcategories: documentation, meetings, release notes and test awareness.

Documentation of tests
Documentation of testing might include both carefully written down test cases as well as reports on which tests have been performed. Stakeholders feel an increased confidence in

tests where the test case is documented in detail, and there is a clear tracking of results for each test case. The confidence is broken down if it is not noted when testing has taken place. The reason, in the word of one respondent: "If you go through all the manual test cases in week 11, things might have changed by week 14. This means you can not trust the results of those tests".

For customers, the documentation of tests are one of the main factors for confidence. A report documenting which tests have been performed and their result can both increase and decrease the confidence of a customer. The proximity to release for delivery of this report matters, a customer feels less confidence in the release if the test report is delivered very close to the release date.

Meetings
In practice, documentation is often complemented and occasionally exchanged for meetings. Particularly managers rely on meetings to establish confidence in the process and verification results. This ties in to the category *people trust*, where a stakeholder using meetings as a confidence source trusts that the release will work as intended if other people do so.

Meetings are also used as a source of knowledge about what is changing for a release, to determine which parts of the system need stricter or less strict quality assurance. Stakeholders who express a feeling of overview about what is changed and what is tested in a release through meetings report higher confidence. A caveat is that this effect decreases with the size of the team. In a team of two persons, all members feel confident that they know about all changes to their product through daily meetings, whereas in a team of 10 persons there is a higher need for documentation.

Release notes
Release notes are a special form of documentation which details exactly which changes are included in the release. During interviews, many respondents specifically noted differences in the quality of release notes between different products. A common comparison was between one product which used a "sluice list" where changes were tagged in version control and verified by stakeholders, and another one where a single person wrote down the release notes during the staging process. Multiple respondents reported much higher overall confidence in the system using the sluice list. They also talked about several incidents relating to lacking release notes. Inclusion in a release of changes which had not been visible to stakeholders and therefore untested was the most common answer to the prompt to describe a time when quality assurance had been lacking. A manager said that they "question the system much more" in reference to the system with poorer release notes, while a customer representative described it as always being tense before a release since something unexpected could be included. The customer also said that "we learned after a while to also test more than what was included in the release document".

A product specialist described the effect of this when saying "if there has been a large change in functionality where you have less knowledge of what has happened, you will not be able to think about what should change in the test cases and not about what has change and what might be affected. That affects the confidence".

A common fear is that functionality that is developed for one customer will be included in the release for another customer. For systems with procedures that do not allow the stakeholders to accurately verify which customers will get which functionality, overall confidence is lower. Multiple product specialists express a need to know which changes are included to which customers to feel confident in the release by verifying if the change is an overall improvement which is acceptable for the other customer or if it includes a change of working which is unacceptable for the other customer. They wish to do a filtering for each new change towards the different customers.

Test awareness
Test awareness is the confidence gained from knowing what is tested and how the tests work. An example is that many product specialists gain confidence from being able to go through the test cases documented in REQ-test which they would be able to replicate and see that they have passed. On the contrary, a common cause for lack of confidence in automated tests is an opaqueness into the test cases from the outside. One respondent gave an example: "It's hard to say how much I trust it. We have detailed test cases, and if the automated testing follows that I might trust it. But I do not know, since I do not know how the automated testing works. For that, I need a developer. For the manual testing it is simple to verify that the customer has the correct balance, got the right shares and received a receipt. Unfortunately, with the automated testing I do not get to verify this myself but just see that the test is green. it is hard for me to say that it gives the same level of confidence."

All respondents who distrusted automated tests for this reason, said that they would trust the tests if they would have insight to exactly what is asserted by the tests. One respondent even said that they would trust automated tests the most if they would be a part of the test creation process, but in absence of that opportunity they would want to review the steps taken by the test. For the automated tests created in efforts purely by developers where the product specialists, managers and customers have not been thoroughly informed about the details of the tests there is no added confidence compared to no testing at all. Several product specialists report that they still manually test parts of the products that are covered by automated tests, since they are not sure about what is tested.

5.1.1.4 Time

While most process categories relate to how something is done, the categories under Time releate to *when* something is done. The same development or QA actions can provide more or less confidence depending on when they are done. Sometimes, the time taken for some parts of the release cycle makes other parts have too little time to be done which lowers confidence in the overall process.

Release proximity
There is a clear relationship between the confidence in a change of the software and the proximity of the time of change to the release date. One respondent even stated that several added features after a set ready for release-date about a month before release date would make them lose confidence completely for the whole release. Several respondents mentioned occasions where functionality was included too late when answering the question about a time when they felt the quality assurance had been lacking. Other respondents stated less severe opinions, but all agreed that the risk for faults caused by a given change is higher the closer to release date the change is introduced. A customer said that the proportion of release content which is done at the ready for release date is a large factor for their confidence in the release. The same customer said that "if something isn't done at ready for release, the bank writes a deviation report and assesses the risk". The release proximity effect goes for both individual changes and for the entire release confidence. If a change is introduced very late, the risk is seen as higher for that specific change to contain faults. If a big proportion of release content is introduced close to the release date, the entire release is trusted less. Respondents noted that when something is introduced late in the process, all acceptance testing up to that point should be invalidated which lowers the confidence in all of the release.

The proximity to release date of when testing documentation and release notes is made available is another factor. One respondent said that when they do not have enough time to process the release notes and plan the testing phase, it lowered confidence in the testing which was subsequently performed.

Lack of time

Even when good quality assurance is planned, the fact that there is a set release date can lead to a situation where there is not enough time to perform all planned activities. One respondent who could not think of a specific time when quality assurance had been lacking answered "Sometimes there was not enough time and you had to prioritize between the test cases. Then you did not feel that everything was completely safe". The same situation was mentioned by several respondents as the trivial case for lacking quality assurance, something that is always a possibility before a release. There is a prioritization made in the test suite where some tests are considered to be minimum and the release can not be trusted if they are not performed. The remaining tests provide progressively more confidence in the release depending on how large proportion of them are performed. This lack of time also existed on the customer side. One customer said that they tried to perform testing of all systems themselves, but when time was lacking they instead trusted the green check-marks on testing protocols. They also mentioned they would like to see detailed test cases, but were not sure if there would be enough time to go through them.

Several product specialists mentioned exploratory testing as important for their confidence but later explained that they rarely have time to perform this kind of testing. Their time would instead go towards performing the manual tests in the existing regression test suite. The same situation seems to exist with test planning. All respondents agreed that the planning phase is the best time for agreeing on tests needed for new functionality and bug fixes as well as for deciding which existing tests had a need for update, but they found that this time rarely existed during that phase and these processes were instead pushed to the testing phase. One respondent described the situation this way: "You should always bring a compromising mindset, you only get limited time and resources to assure the quality both through tests and maybe through corrections and development so it is a bit of give and take. You are rarely satisfied, but instead usually have to agree on what is good enough. At least when it comes to planning of new development".

Stress

The final time factor is stress. When there is less than normal time available to do something, the confidence decreases even if the time suffices. One developer said that "fixes introduced in a late stage gives space for worry. It is the human factor when people are stressing. It is easy to become speed-blinded and miss obvious deficiencies at that stage. The risk is high. That is when you can feel worried, when changes are pushed through quickly. At that point, you can almost forebode that there will be a need for additional measures after release." Other respondents pointed out that both the time available for planning and development are important to avoid stressed-out solutions. The reasons given were that it gives time for multiple people to take part in the process and prevents the solution from being a result of stress.

5.1.1.5 Trust

Sometimes, confidence arises simply because of trust from the stakeholder. In certain situations they do not feel any need to inspect the process or verify hard data to ensure the quality. These situations are typically when they trust the people involved or when there is a history of proven results for a process and/or system.

People trust

People trust is especially common in the manager and customer categories of respondents. They sometimes rely completely on one or multiple other persons for their confidence in a release candidate, by not doing any quality assurance activities on their own. For managers and customers, they trust the judgement of product specialists for providing confidence in the release. Product specialists on the other hand, sometimes trust developers. This is the case with some kind of automated tests, where they simply read the results from the auto-

mated test runs and trust the developers to have implemented sufficient tests. The product specialist are not always comfortable with this however, since they have the responsibility for quality there is a wish to not have to rely on this trust. Developers in their turn trust the verdict from product specialists and customers, when they deem a release candidate to have high enough quality the developer tends to have confidence in the candidate as well.

Another factor of people trust is in the development work. Development done by an experienced developer with a good track record of fault-free code and good knowledge of the system is given higher confidence. A product specialist also said that they put less time in testing for a change delivered by a senior developer than the same change done by a junior developer.

Proven results
All systems and quality assurance processes increase in the amount of confidence they provide after time has passed and their results have been proven to be good. The factors mentioned to be trusted more over time are:

- Automated test suites becomes more trusted by all stakeholders over time when they are proven to find bugs and no additional bugs are found after a passing test suite.
- General quality of a new product becomes more trusted by customers after several consecutive successful releases.
- The release schedule with quality assurance activities becomes more trusted by all stakeholders after several consecutive releases when no need has been found to add any activities to the schedule
- Test environments become more trusted after several consecutive releases where no difference has been found in functionality of the product in the test environment and in the production environment.

5.1.2 Verification Results

The second main category is Verification Results. It relates all categories where stakeholders use the results from verification of hard data, such as test results or facts about the software itself, to establish confidence in the release candidate. Verification results are what most people first think of when it comes to quality assurance. Categories under this main theory are mostly about functional testing, but there are also categories relating to factors such as compliance to regulations and knowledge about the test environments. A summary of stated situations in interviews of where different kinds of testing provides the most confidence is seen in Table 5.8.

5.1.2.1 Automated or manual testing

As noted in other section of this thesis, there is a big debate about the relative importance of automated and manual testing in continuous delivery. This debate applies also to the question of the level of confidence placed in the tests, where respondents see factors increasing and decreasing their confidence in both manual and automated tests. They use the different kinds of tests for different purposes, and all agree there is still a need for both kinds of testing in a continuous delivery context to establish sufficient confidence in the quality.

Manual testing
Manual testing is done with several sets of test cases which include

1. A set of standard regression test cases which are always done.

Code	Found in interviews	Total references
Complexity of change	5	5
Kind of change	4	12
Magnitude of change	5	7
Creation, updating and selection of regression tests	5	11
Requirement specification	3	9
Testing responsibility	3	3
Documentation of tests	5	8
Meetings	2	3
Release notes	6	20
Test awareness	7	19
Lack of time	9	15
Release proximity	7	17
Stress	2	3
People trust	8	13
Proven results	5	10
Ad-hoc testing	4	6
Automated testing	11	42
Desired coverage	2	4
Manual testing	11	36
Environments	3	4
Dependencies	2	4
Third party integration	4	4
Compliance	1	1
Look and feel	2	2
Performance	4	5
Security	3	6
Small faults	3	3
Test cases	8	19
Unexpected changes	7	13

Table 5.7: Number of references for each code found in open coding

Situation	Most suitable testing kind
Process utilizing external services and systems	Manual
Process spanning multiple days	Manual
Regression testing of generally stable systems	Automated
New functionality	Both manual and equivalent automated for the first release, automated for subsequent releases
Verifying look and feel	Manual
Testing systems in a state of rapid change	Manual
Verifying versions of third-party dependencies	Automated
Security testing of critical systems	Manual
Security testing of non-critical systems	Automated
Performance	Mostly automated, but manual for new or problematic systems

Table 5.8: Respondent notes on suitability of manual and automated testing practices in certain situations

2. A set of regression test cases which might be picked depending on perceived need.

3. Exploratory testing based on the changes introduced to the release.

Cases from 1 and 2 are stored in the REQ-test tool and their results are documented in the same to provide confidence for all stakeholders. Cases in 3 are usually performed ad-hoc and are intended to uncover bugs and provide confidence only for the performers of the testing themselves. A typical manual test might be quite complicated: involving several systems, each with many facts asserted and spanning multiple days to emulate real life behavior of e.g. a transaction.

The reported pros of manual testing is that it is easier to spot faults in pieces which are not explicitly searched for in the test case. As a concrete example, one respondent said that they might be testing a change in the stock trading functionality, but during testing notice that something looks different in the derivative part of the window. These kind of discoveries comes naturally for a human eye but would require substantial dedicated effort to ensure coverage in a completely automated fashion. This fact results in stakeholders having slightly less confidence in a system which has not had any manual testing, since they have to trust that the automated tests cover all expected and unexpected changes. It is also easier to adapt the testing when test cases are manual. If the system has changed slightly to require a test to be performed differently there is usually little or no extra work for the manual tester, compared to automated tests which might break and require rewriting. This makes manual testing more suitable than fully automated testing for systems which are in a state of rapid change, while the advantages drop off when the system stabilizes to allow tests to be run in identical manners each time.

Some aspects of manual testing lowers the respondents confidence in the results. One aspect is the human factor, as one respondent put it: "There is always a risk [with manual testing], it may happen that you lose focus, that someone just has to ask a question and you forget where you were in the testing process". This lowers the confidence in manual tests especially for other stakeholders than the one performing the tests, who can not verify exactly which steps have been performed to attain the test results. Another problem is the time consumption, which frequently results in the manual test suites not being fully performed as they grow to include more test cases. This is in fact the most common reason stated in this study for distrust in manual test suites, that while the test suite itself is enough to provide full confidence there is rarely enough time to work through the entire suite. A continuous delivery context makes this issue even larger, since it makes long periods of time to perform extensive manual test suites impossible.

The last issue which lowers confidence is the time of performing. Since manual regression tests are commonly only performed once before a release, there might be changes appearing after the tests have been performed which invalidate the results. The solution would be to redo all tests, but this is hindered by the previous point of great time consumption from the product specialists who need to perform other tasks as well before a release.

Automated testing

Automated testing covers all stages of software quality assurance. It includes the lower levels of testing such as unit and integration testing, but in this discussion it most commonly refers to system level testing and automated acceptance testing using tools such as Selenium and Cypress. The automated tests can be created solely by developers, solely by product specialists or by developers but guided by product specialists. There is a difference in the kind of tests which provide confidence for different stakeholders. Customers, managers and product specialists mainly trust automated acceptance tests while developers place higher trust in unit- and integration tests.

Interviewed stakeholders agree that a higher level of test automation gives a better confidence. The main reason is that it ensures that a sufficient number of tests can still be run in

the reduced testing time that a continuous delivery context brings. The quick and effortless run of tests also allows it to be run multiple times, to enable the entire test suite to be run after every change of the software. The rise in confidence applies to all categories of stakeholders, including customers who may not necessarily see the tests or their results but report increased confidence from the knowledge that a system is covered by automated tests.

Respondents have consistently approximated the ratio of test cases in the currently existing regression test suite to be replaceable with automated tests to be 70-90%. They report that performing automation of this percentage would allow product specialists to focus their efforts on the remaining test cases which would allow for the entire regression test suite to be done in every release, effectively increasing overall confidence.

As discussed in section 5.1.1, the confidence placed in automated tests depend largely on how they are implemented. A common reason for a lack of confidence in a release candidate despite passing tests is the lack of knowledge about what is tested. When knowledge was present, there could still be issues if the quality of the test suite was not perceived to be high enough or it had historically missed bugs in the software.

Ad-hoc testing
In addition to the planned and structured testing, some respondents felt they could not fully trust the release candidate unless they had a chance to do ad-hoc testing in the end of the testing process. They wanted to go through what had been tested and do a few extra test cases based on hunches and feelings.

5.1.2.2 Environments

To trust the results from testing, stakeholders require the test environments to be of sufficient quality. One respondent said that "There is very varying quality of the data in some test environments. It is a lottery to know if the results of tests are correct in those environments. If they pass they tend to be correct but when they fail there are many parameters that can play into why it went wrong." This brings similar confidence issues as the breaking tests discussed in the previous section.

5.1.2.3 Test cases

Apart from how the tests are performed, stakeholders generally place highest importance in the test cases themselves. This means that they require test cases to sufficiently cover scenarios which they deem important for the functionality of the product. They also feel a need to assert that the existing test cases are enough to discover a fault. Several respondents said that they want to see that "something becomes red" if they try breaking a part of the software.

5.1.2.4 External

The quality assurance against parts of the system which were not developed by the company itself were handled in separate ways from the software developed in-house. Stakeholders found the presence of external pieces to be problematic both for its primary impact on confidence and for the impact it had on testability.

Third-party integration
Services which are tightly integrated to third party systems tend to receive less confidence. The reason for this is mainly that test environments become unrealistic. An example reported is market connection, where the test market is not guaranteed to behave the same as the real market which makes confidence in the testing done on systems depending on the market integration lower. In some third party testing integrations there was also problems with resetting testing entities, which resulted in an inability to introduce automated tests. Since a

higher percentage of automation heightened the confidence, this has a negative impact.

Dependencies
Third party dependencies are seen as a risk. Since they can not be tested in the normal way but instead have to be trusted to work, the presence of external dependencies adds possibilities for both general software faults and for security vulnerabilities. The main worry about third party dependencies are security vulnerabilities, since there is a high risk of any particular vulnerability in a third-party module being discovered by a threat actor and exploited in all systems depending the module. There is also a concern of licensing, where careless inclusion of any dependency might cause the software to include licenses not compatible with the intended usage. There is more confidence placed in the fault-freedom of a release of software containing fewer third party dependencies than in one containing more. Stakeholders feel more confidence in the release when they know that they are using the latest version of a dependency where no bugs or vulnerabilities have been found.

5.1.2.5 Non-functional

While most verification is done of the functional requirements, several non-functional factors have been raised by respondents as important for the overall confidence of the release candidate. These include *compliance*, *look and feel*, *performance* and *security*.

Compliance
For those in charge of verifying compliance, it is a strict requirement for letting a release candidate move to deployment. The category includes both compliance to regulatory demands and to customer agreements. Compliance is usually stated as a set of criteria which needs to be fulfilled, and when any of the criterion is not fulfilled there is no confidence in the candidate while there is room for confidence if all are fulfilled. The stakeholders in charge of compliance feel confident when they can verify compliance is fulfilled. The criteria for most regulatory and customer agreements include regular security tests at given intervals.

Security
Similarly to compliance, security is a strict requirement in the sense that all confidence is lost in the release candidate if a vulnerability is found to be present. It is different since there is not a list of easily verifiable criteria, but instead there is a defined secure software development lifecycle where the confidence in security level increases with the rigor of each step. The secure software development lifecycle as defined for Crosskey can be seen in Figure 5.3

Performance
In some cases, the verification of performance is important for stakeholders. However, the respondents saw it as something they worry about for new systems and in cases where bad performance was an issue in the previous release. In those cases, they needed to see dedicated testing of performance done in a very production-like environment. In other cases, all respondents were satisfied with trusting that performance had not degraded from the latest release as long as the automated tests passed their timeouts.

Look and feel
Look and feel was discussed by respondents to be something which was not explicitly checked, but could marginally lower the overall confidence if it was poor. Product specialists and customers reported that while it would not stop a release, they always noted if elements were not in logical positions or if pages do not look okay. To trust that this was the case, they wanted to manually take a decision for all new or changed pages.

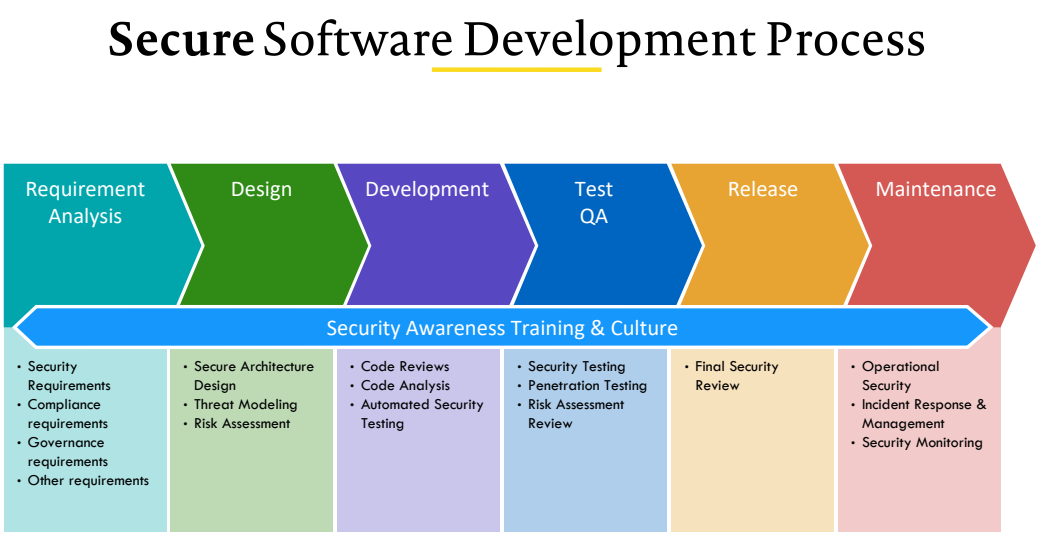


Figure 5.3: The Secure Software Development Lifecycle for Crosskey

5.1.2.6 Attainable verification level

Even with all existing verification, respondents note that the goal is not to verify 100% of the quality expectations which is seen as impossible. There is always a perceived possibility for unexpected changes or errors, and the overall confidence of a release may still be high enough even when there are known errors.

Unexpected changes
As previously mentioned, one respondents summarized their view of release confidence as "you were always tense before a release, since something unexpected could be included". Even with rigorous testing, they noted that there was room for unintended changes to the system where "no one had thought that this thing could affect this other thing" and therefore no testing was in place.

The perceived risk of unexpected faults or other changes was especially high for changes in the system coming from other departments withing the organization, where insight into the quality assurance procedure was not as high as for their own department.

Tolerance for small faults
Respondents said that in some cases, the confidence for a release candidate can be seen as high enough even when faults have been found. The criteria for a fault to not stop a release was that it had to be present in a scenario that rarely occurs in production or affects very few customers. The intended functionality which broke also had to be deemed to be replaceable by manual interventions or other kinds of workarounds. Additionally, a fault in the look and feel could be seen as insignificant enough to not affect the release decision.

Desired automated coverage
It is not desired by any of the stakeholders to have an automated test suite covering 100% of the software with all possible scenarios and they do not require it to feel confident in the release candidate. One respondent said "I do not think anyone has 100% coverage, that is too extreme", while still being of the opinion that "It is always motivated to add as much tests as

possible. Not just for the sake of it, but when you do changes somewhere else in the system you do not get unexpected breaking changes".

Respondents have found that a too-high coverage does in fact decrease coverage. The reason for this is the fact that changes to the software will make tests break, the tests will fail even though the functionality still works as intended. One respondent who was in the process of recreating a suite of Selenium tests in Cypress commented on this: "There [in Selenium] we have tests which go through everything. Now we are thinking about how we want to have it. Currently in Cypress we only do the happy-case and see that all scenarios work in normal cases. [...] With the Selenium suite there were tests breaking every release. You only had to switch names on a field or something and then you had to fix the test." The result of this was that unless the organization allocated unusually large resources to test maintenance the test suite never showed fully positive results which decreased overall confidence in the suite.

5.2 Literature Study of quality measurements

This section presents the results of the literature study described in section 4.3. The effectiveness and potential usages of each method is described in this section, for explanations of the different measurements and quality assurance practices see section 2.2.

5.2.1 Software metrics

From the categories of metrics enumerated in 2.2.1, the papers reviewed by Radjenovic´ et. al. [41] found process metrics to be the most effective for fault prediction, followed by object-oriented metrics and lastly traditional metrics.

5.2.1.1 Traditional metrics

Regarding size metrics, a higher number of LOC has been shown to be correlated with increased fault-proneness of the measured software [41]. The increased difficulty of comprehension, development and testing of software caused by a high complexity has been shown to lead to a higher number of faults in the finished product, and also to a higher number of security vulnerabilities [29, 14]. As a complexity measurement, the commonly used McCabe´s cyclomatic complexity measure is a good predictor of fault-proneness in large projects using object oriented languages [41].

5.2.1.2 Object-oriented metrics

In the CK suite described in Table 2.1, the different metrics have been found to have differing levels of effectiveness for fault prediction [41]. Table 5.9 shows the results of effectiveness in the different CK metrics.

Metric	Performance
WMC	++
CBO	++
RFC	++
LCOM	0
DIT	0
NOC	0

Table 5.9: The effectiveness of CK metrics as found in [41] in the scale [–, -, 0, +, ++]

5.2.1.3 Process metrics

The age, number of changes and change set metrics have been found to be effective for predicting faults in a software release, while there is mixed data on the performance of developer and past fault metrics [41].

An important benefit of using process metrics over static metrics is the false positive factor of many static metrics. As Moser et al. [36] states, a complex file can be fault free even though it is classified as fault-prone based on the metrics since the developer may have done a very thorough job in coding and avoiding errors. However, with a lot of changes the probability that one of those changes introduce a fault is high. Similarly, a complex module that has been observed to work well for a long time might be considered fault-free, while a complete rewrite of the module can be expected to introduce faults even if the resulting complexity is the same as it was before the rewrite. Moser et al. found process metrics to be more efficient than static code metrics, and the conclusion is shared by several other studies who find that process metrics are superior in finding post-release faults compared to source code metrics [41].

5.2.2 Code smells

Code smells overall have been proven to indicate an increased probability of bugs in the finished software [3]. However, this is not universal for all code smells. For instance, the code smell *Switch statements* which says that a high number of switch statements in the code is a smell, have been found to not be connected to increased fault-proneness of the software [22]. Other code smells have been found to have mixed results, only provide small increase in faults or even reduce faults in some cases. These include Data Clumps, Speculative Generality, Message Chains, and Middle Man [22].

Tools such as Checkstyle [10] and PMD [40] using metric-based detection of code smells have shown good results in detecting a number of code smells [34].

5.2.3 Testing

This section describes the results found on the testing-related categories.

5.2.3.1 Automated or manual

It is sometimes thought that a CD pipeline must be completely freed from manual testing. However in a study with 25 interviews on test activities in the continuous delivery pipeline, ten of 17 interviewees talking about automated testing described manual test activities as "an important complement to automated testing" [32]. They found that some stakeholder interests are better served by automated testing while some are better served by manual testing. Automated testing was found to be best in cases where the test activities are repeated, for their classified stakeholder interests *Secure stability, Check changes* and *Measure progress*. An interesting finding was the need for exploratory testing, about which they stated that "Whereas automated test activities in the pipeline are able to rapidly provide feedback to developers and to verify requirements, exploratory testing can provide more in-depth insights about the system under test." This further adds to the view that a combination of manual and automated tests is a desired approach.

Rafi et. al. found supporting views in a structured literature review where they evaluated the benefits and limitations of automated testing in contrast to manual testing. They found the benefits of automated testing to include *Improved product quality* with fewer defects present, *High test coverage* which allowed for extensive regression testing and reduction in reduction in cost, testing time and human effort [42]. They also found an increase in confidence in the quality of the system as perceived by developers. The most important limitation found was that "Not all testing tasks can be easily automated, especially those that require

extensive knowledge in a domain." In the same paper, a survey was performed with industry practitioners. Among 115 answers, 84,35% were satisfied or Highly satisfied with the use of automated testing practices, but still the statement "Automated testing fully replaces manual testing" was rejected by 80% of the respondents. The general view was that automated testing should be used as a tool to make the work of testers more efficient and allow them to work on tasks where they provide more value.

Same as in general quality assurance, the question of automated or manual testing comes up in security testing. Similarly to quality testing there is a view that the preferable approach is a combination of manual and automated testing as shown in a paper by Sing et. al. [49]. Automated testing is seen to speed up the checking of common vulnerabilities and scanning through large numbers of locations to check for sensitive data. However, there is still a need for manual penetration testing to find unique vulnerabilities.

5.2.3.2 Unit- and integration testing

The trade-off between unit testing and integration testing is one of run speed to confidence and maintenance. Integration tests typically take more time to run since actual services need to be used, but since they do not rely on mocked objects having the same implementation we can be more confident that a passed test means that the code will work in reality while at the same time reducing the burden of maintaining mocks [45].

5.2.4 Test quality

This subsection describes the results found on effectiveness for different ways of measuring test quality.

5.2.4.1 Test coverage

It has been established that the metric of test coverage is not enough to provide a guarantee of test suite effectiveness, in a paper by L. Inozemtseva and R. Holmes [24]. This paper evaluates the relationship between test suite size, coverage, and effectiveness for large Java programs. They conclude that "coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness". It has also been found that developers do not have confidence in a test suite to prove the code is bug free when code coverage is the only test quality measure [6].

5.2.4.2 Mutation testing

Mutation testing has been found to accurately identify weaknesses in test suites and measure test quality. It has outperformed test coverage measures in finding issues in tests, as well as been able to find issues in test cases which are too obscure to be detected by manual review [5].

5.2.5 Code review

In addition to detecting functional faults, it has been found that code review can identify common types of security vulnerabilities [8].

Since code review is a manual and subjective process, the ability of a reviewer to detect faults depends on several factors and can be negatively affected by the number of directories under review, the number of total reviews by a developer, and the total number of prior commits for the file under review [39]. This means that a review becomes less effective at detecting vulnerabilities when performed by a developer who has a big review burden or when the changes are big or complex.

5.3 Expressing confidence level

To state that confidence has been established for a release candidate, each factor should be satisfied. For some of the factors, it is not feasible to try expressing them in a continuous delivery pipeline. An example of this is the factors belonging to the category Information Availability (5.1.1.3). For other factors, the quality measurements presented in section 5.2 can be used as a data source to determine the confidence level. As suggested by Ahmad et al. [2], the ultimate decision on whether there is confidence in the release candidate is still a matter of informed opinion based on this data.

In Table 5.10, examined quality measurements suitable for specific confidence factors are presented. Factors excluded from this table are considered by the researcher to be unfit for expression in a pipeline based on the research done for this study.

Each confidence factor in Table 5.10 belongs to a category from section 5.1. The subsection number for the category is mentioned in the first factor, each subsequent row belongs to the same category until a new subsection number is mentioned. The source of data is either a measurement from section 5.2 or a description on how this could be achieved manually based on how it is currently done by the interviewed stakeholders.

Table 5.10: Suggested data sources to display in the pipeline to satisfy confidence factors, with motivation for the choice.

Confidence Factor	Source of data to pipeline	Motivation
5.1.1.1: Type of change	Automated metrics gathering, some manual entry. Detailed below.	
Magnitude of change	Code delta and code churn (2.2.1.3) of LOC (2.2.1.1).	As described in 5.2.1.3, the combination of code delta and code churn gives a good view of how much a metric has changed. LOC is the standard metric for size and has been correlated with increased fault-proneness (5.2.1.1).
Complexity of change	Code delta and code churn (2.2.1.3) of McCabe´s cyclomatic complexity (2.2.1.1) , WMC, CBO and RFC (2.2.1.2).	As described in 5.2.1.3, the combination of code delta and code churn gives a good view of how much a metric has changed. McCabe´s cyclomatic complexity is the standard metric for complexity, and WMC, CBO and RFC have shown good effectiveness for fault prediction (5.2.1.2).
Kind of change	The system affected by the change can be automatically fetched from version control, while the others have to be manually entered. This can be done e.g. in an issue tracker or with prefixes in branch names or commit messages.	While the system is directly correlated to where a specific commit has been made, no reliable metrics have been found for the other types enumerated for *kind of change*.
5.1.1.2: Following defined steps	Manual entry. For each step in the category, the relevant stakeholder has to note whether it is fulfilled or not (binary yes/no entry). This can be done e.g. in issue tracking.	There is no way to automatically check whether these steps have been taken, yet they are important for confidence. If tracked in issue tracker, it is known whether the steps have been done for a particular issue included in the release candidate.

5.1.1.4: Time	Irrelevant in a continuous context	Without a release date, there is no information on the time factors available.
5.1.1.5: Trust	Measuring historical data about the system and developers. Code review.	
People trust	Check whether a code review (2.2.5) has been performed. Use developer metrics (2.2.1.3).	Code review has been shown to effectively detect faults and security vulnerabilities introduced in a change, thus eliminating the need to trust a single developer to not introduce them. Developer metrics can be used to affect the confidence level based on the developer committing the change, but have shown mixed results in effectiveness which suggests caution in how much weight it is given.
Proven results	Measure the number of releases passing the criteria enumerated under *proven results* in section 5.1.1.5 and increase score for each pass	For each successful release for each criteria, the confidence for the correctness of the entity (system, test suite, etc.) concerned rises.
5.1.2.1: Automated or manual testing	Results from automated or manual testing according to the reported results in Table 5.8.	The test results are the most important metric, after being influenced by results from other metrics they depend on.
Manual testing	Manual test results recorded in a test management tool.	
Automated testing	Automated unit tests(2.2.3.2), integration tests (2.2.3.3), system tests (2.2.3.4) and acceptance tests (2.2.3.5). Security scanning. Measurement of traditional (2.2.1.1) and object-oriented metrics (2.2.1.2), as well as detection of code smells (2.2.2).	All of these methods have been proven to reliably detect faults and/or indicate quality levels in the software.
5.1.2.2: Environments	No good data source found in the scope of this study, future work should find suitable data sources.	
5.1.2.3: Test cases	Mutation testing (2.2.4.2) and code review (2.2.5) of tests.	Mutation testing can accurately identify weaknesses in test suites. A code review additionally heightens the confidence in that the correct things are tested.
5.1.2.4: External	Automatic scanning and manual entry	
Dependencies	Dependencies can be monitored with automatic scanners detecting outdated versions.	
Third party integration	The presence of a third party integration of a system has to be manually entered.	
5.1.2.5: Non-functional	Varied, detailed below.	
Compliance	Manual entry	

Security	Manual entry of requirement analysis, then either automated scanning or both automated scanning and manual testing based on analysis results. Code review.	See Figure 5.3.
Performance	Assumed no extra verification needed, unless manually entered need in which case manually entered results are needed	Performance was usually seen as included in normal automated tests, and only in special cases needed to be done explicitly.
Look and feel	Manual entry	Stakeholders did not trust any automated ways to measure look and feel, and required a person to have looked at it before release

6 Discussion

This chapter contains a discussion by the author of the results and method of the thesis.

6.1 Results

In this section, the results of the different parts are discussed. Interesting and divergent results are lifted for analysis and discussion.

6.1.1 Relative importance of the main themes

Many confidence factors were found in the Process-theme. This is interesting, since the bulk of quality assurance work, both practical tasks and research about quality assurance, lies in testing which belongs to the Verification results-category. Respondents were clear that the source of truth usually lies in the verification results, but process factors influenced how much confidence was attached to a positive result. It should be noted that there might also be a possibility that process factors were mentioned more simply because they are less established, and a good base for verification results is already in place which leads to the respondents not thinking much about it.

6.1.2 Relative importance of subcategories

It should be noted that all categories do not have the same weight in the final decision of whether to release a candidate or not. This is mentioned by stakeholders under the heading Tolerance for small faults in section 5.1.2.6. For instance, a failure to fulfill compliance criteria would make it impossible to take the release to production even if all the other confidence factors are fulfilled. Meanwhile, a failure to check look and feel would not disqualify the release and it could even go to production with known faults in this area. The importance of the compliance category motivates the choice to keep the factor even though it was only referenced by one respondent.

6.1.3 Measurable and immeasurable factors

Some views on continuous delivery are based on the assumption that quality can be measured and quantified. While this is true to a certain degree, this thesis challenges the thought

that confidence can be established solely from the results of tests. Many other factors need to be considered for the test results to say something about the actual quality of the software. An example is the category Information availability (5.1.1.3), which shows how stakeholders can lose confidence in a release with perfect quality scores if there is no insight into how these scores were measured. The information availability itself, however, is not possible to measure.

6.1.4 Expressibility in pipeline

As shown in Table 5.10, many of the confidence factors are not directly quantifiable or take significant effort to quantify. In addition to this, many of the confidence factors described in section 5.1 are more related to underlying organizational factors than factors differing from time to time. This does not mean that the confidence factors are impossible to satisfy in a continuous context, but it indicates that expressing a general confidence level in a pipeline might not be the best way to retain quality assurance when moving to continuous delivery.

6.1.5 Suggested alternative solution

The results indicate more towards a solution of satisfying some confidence factors (such as those mentioned in section 5.1.1.3: Information availability) through organizational changes. The measurable results from Table 5.10 should still be included to get a picture of the predicted fault-proneness of the change, but the manually entered ones might slow down the process more than if they are assumed to be done, without adding additional quality guarantees if they are seen as a chore by the stakeholders. This would hinder, rather than help, the move towards shorter release cycles.

6.2 Method

This section discusses and criticizes the method. Potential risks and shortcomings of the study are discussed and their potential consequences for the results are analyzed.

6.2.1 Grounded theory approach to confidence factors

The validity of the study is supported by coding all interview data, leaving out only side discussions not related to the subject. This combined with ensuring that the respondents have shared all factors they consciously use to establish confidence ensures that the analysis included all factors possible to include from the data.

6.2.1.1 Literature review before data collection

In classic grounded theory, no literature should be consulted before the theory is emerging and the researcher should start with a clean theoretical slate. In Straussian grounded theory, it is seen as acceptable to consult literature throughout the process [50]. In this study, the literature review on quality metrics was done before the grounded theory study. This allowed the researcher to have better knowledge of which methods could be used to acquire confidence, heightening the theoretical sensitivity by having knowledge of surrounding areas while not influencing the clean slate of emerging theory by performing a literature study in the direct subject of study. The positive influence of performing literature studies on subjects adjacent to the subject studied in GT has been described by Thistoll et al. [51].

6.2.1.2 Bias risk

With qualitative research, there is an inherent subjective component which makes the results more prone to bias than in quantitative research. Seeing that the main result of this thesis are

from analysis done by the researcher, it is likely that it would be interpreted slightly differently if performed by another researcher. To combat this, all results are correlated directly to transcripts of the interviews which makes it verifiable and grounded. If another researcher would have similar background knowledge, the reliability of the analysis process should be relatively high and produce the same categories. Many qualitative studies attempts to lessen the subjective impact of a researcher by having one researcher perform the coding and another researcher verify and criticize the codes. Some even perform blind coding where several researchers code the same data and then compare the codes for similarities and differences. This study was limited by having only one researcher, which heightens the risk for subjective bias introduced in the coding.

6.2.1.3 Left out theoretical sampling

There was a theoretical sampling performed in addition to the interviews and the security presentation, which consisted of records exported from the test management tool REQ-test. The intent was to find ways in which confidence was defined through the analysis of test cases since the specific test cases were mentioned as sources of confidence. The records were not included in the final analysis, since the results were found to not converge and provided little information of value.

6.2.1.4 Risk of information loss in translation

The interviews were performed in Swedish and transcribed in Swedish. The analysis and formation of categories were performed in English, and direct citations were translated by the researcher who is not a professional translator. This leaves a risk for information loss and misunderstandings in the translations, which could influence the meaning and interpretation of respondent statements included in categories. This risk was lessened by double-checking important translations with machine translation.

6.2.1.5 Subject selection

As stated in the delimitations of the study, the respondents were chosen from the capital markets department. This allowed for a good view of what confidence means for the stakeholders who are a part of this department. However, some respondents mentioned interest from other departments in a release of capital markets products. An example is performance testing, which most respondents did not consider necessary but some mentioned that the Infrastructure and operations department might do performance tests of capital market products before release. This implies that there might be additional factors in other departments which need to be satisfied, which could be found in an extended future study involving larger parts of the organization.

6.2.2 Literature review

The literature review was limited by focusing on established results already appearing in an existing systematic literature review. This excluded newer results which might prove to be more efficient than the existing measurements.

6.3 Source Criticism

The literature used in this thesis is mainly peer-reviewed journal articles and conference proceedings. The sources included in the literature review were mostly secondary sources included in the tertiary review by Champion et al. [9] who included only peer-reviewed articles and explicitly removed items published through predatory publishers. It should be noted that the tertiary review itself was a preprint at the time of this study, so measures were

taken to verify the claims of the secondary sources. For the sources included in the literature review through search, care was taken to apply the same quality criteria.

Some theory on quality measurements and method was based on textbooks. The textbooks are well cited and from established authors in their fields, and are therefore considered to be reliable sources. A single use of a blog post exists to explain the practical trade-offs between unit- and integration testing. This is considered the least reputable source but was published in the engineering blog of Spotify which is an established technology company which gives the arguments some credibility.

6.4 The work in a wider context

This section reviews the societal and ethical aspects of this work in a wider context.

6.4.1 Economic and societal aspects

The usage of the results from this thesis can have a positive societal impact by allowing the time and effort needed to assert confidence in a release candidate to go down, which allows the organization to move towards shorter release cycles. This is positive for the company since there are less resources needed for quality assurance, resulting in a better economical situation. The staff of the company can better focus their quality assurance efforts when they are aware of which factors to work towards which would allow more staff members to feel they are doing useful work as well as allowing them better career opportunities based on others experiencing the value of their work. Having more confidence together with a faster release model could even result in the company increasing profits as a consequence of accepting new projects they otherwise would not have accepted since they know it can be done reliably and efficiently.

Similarly, it is also positive for the customers, since they are able to order more functionality and there is less time between ordering and delivery of functionality. Having increased confidence in the quality of the delivered product will cause less resistance to ordering new projects and allow more room for incremental improvements of their product.

There is a positive impact for all involved parties from using carefully researched quality measurement, because of the lesser number of faults in the finished software. This is especially beneficent for end users, who obtain a lesser risk of disruption in their day-to-day usage of the software. Since the products created by Crosskey are banks that are critical infrastructure, the improved quality impacts society as a whole by avoiding banking outages which could hinder daily life by not allowing people to pay for what they need. The increased incremental improvements discussed earlier in this section has a positive societal impact by creating better banking systems to be used by end users.

There are also negative impacts. Doing more quality assurance in continuous delivery requires more infrastructure since tests and metric collections are run more often. Some of the suggested methods such as mutation testing require very large computational resources to work on large code bases. This causes increased consumption of hardware and energy, which has a negative economic impact for the company as well as a negative environmental impact.

6.4.2 Ethical aspects

Removing manual effort placed in quality assurance carries an ethical risk, which stems from replacing humans with automation. If enough confidence factors are eventually satisfied through automated means, the risk is that the people checking these factors today could lose their employment. However, this risk is seen as low because of two factors:

- Multiple confidence factors require human judgment and no automated measurements are anticipated for these in the near future

- The product specialists have reported being overworked rather than underworked today, and expressed a want to focus on more important tasks than manual repeatable quality assurance.

This makes it likely that the effect would be a more valuable output from the human staff, rather than a reduction of staff.

There is a risk of over-reliance on standardized and automated processes for confidence establishment. There might be severe faults released in the software which would have been easily discovered with exploratory testing but missed in the new process since they are outside of the regular confidence factors. In the context of banking, this could cause massive economical damage with only the process to blame.

7 Conclusion

The purpose of this thesis was to explore the definition of confidence in the quality of a release candidate, specifically which factors stakeholders use to establish confidence. Additionally, it was to map established quality measurements to confidence factors to express the confidence level of a release candidate in the continuous delivery pipeline. The results are a good view of the requirements for confidence in a release candidate from stakeholders at the Capital Markets department at Crosskey. This is useful for all parties involved in transitioning from traditional release handling to shorter release cycles and ultimately continuous delivery, as it is important to ensure these factors are satisfied either with the metrics presented in this thesis or with alternative methods.

7.1 Research Questions

In section 1.3, three research questions were presented. These questions are answered below, based on the results presented in chapter 5 and discussion in 6.1.

Which factors do stakeholders in an organization take into account in order to establish confidence for a release candidate?
Stakeholders at the Capital markets department at Crosskey consider both process factors and verification results to establish confidence for a release candidate. They rely on verification results to ensure the quality level and attach differing amounts of confidence to the results depending on how well the process factors are fulfilled. The complete set of confidence factors are described in section 5.1.

Which quality measurements are most effective in a continuous delivery pipeline to establish confidence in a release candidate?
Quality measurements can be split into metrics that can be measured directly and testing which has to be planned for the specific use case and actively performed. Among metrics, process metrics are the most effective, followed by object-oriented and lastly traditional metrics. Internally, these categories have specific metrics with varying effectiveness described in section 5.2.1. In testing, neither automated nor manual testing can be said to be universally superior. Instead, each kind of testing have their respective areas where they are a better fit, outlined in section 5.2.3.1 and with specific results for Crosskey presented in table 5.8. The

confidence in tests themselves is not sufficiently measured by coverage alone. It can be better ensured by a combination of mutation testing and code review of tests. In addition to the mentioned measurements, measuring the number of code smells present and the presence of code reviews can also improve confidence in a release candidate.

How can the confidence level of a release candidate be expressed in a continuous delivery pipeline to stakeholders?
Expressing confidence in a release candidate can currently not be done effectively and comprehensively with quality measurements alone. Several confidence factors, mainly from the process category, rely either in large amounts on manual inputs or are unsuitable to be expressed at all in a pipeline with the quality measurements reviewed by the examined literature reviews. The factors not fit for direct expression are mostly static in the organization, meaning that a general knowledge that they are always fulfilled can be achieved. If this is known and can be trusted, the measurements described in table 5.10 can be used in a pipeline to express the confidence level.

7.2 Consequences of the work

The motivation as presented in section 1.1 was to find how an organization can ensure stakeholders have the same confidence in release candidates when moving to shorter release cycles and ultimately continuous delivery, as they have when using traditional release management. To fulfill this goal, the organization must ensure that the confidence factors presented in this thesis are satisfied. This should be done by a combination of established processes known by all stakeholders and measurement results expressed in the pipeline for the release candidate.

7.3 Future work

The confidence factors found in this thesis are the result of a qualitative study. Qualitative methods generate theory and produce hypotheses. The population size in this study is 11 which is not the full set of Capital Markets stakeholders, only a small proportion of the size of the entire organization and very small compared to the industry at large. The statement that these results comprehensively cover the factors stakeholders use to establish confidence in a release candidate, in this organization or any other organization, should therefore at this point be seen as a theory and not an indisputable fact. To verify this theory, a quantitative approach involving a survey for the usage of these confidence factors and for whether they are enough for all stakeholders in the implementing organization could be performed.

If a wish for expressing a larger proportion of the confidence factors directly in the pipeline using quality measurements exists, this study can be used as a basis for a more thorough exploration of available measurements. This study was constricted by including only measurements that have already been covered by several studies included in existing literature reviews. There may be more recent and/or less studied measurements that could provide data for specific confidence factors. A researcher interested in quality measurements can use this study to determine confidence factors not yet mapped by quality measurements and test new approaches in order to provide data to satisfy these factors in a more automated fashion.

Bibliography

[1] William Adams. *Conducting Semi-Structured Interviews*. Aug. 2015. DOI: 10.1002/9781119171386.ch19.

[2] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. "Data visualisation in continuous integration and delivery: Information needs, challenges, and recommendations". In: *IET Software* 16.3 (2022), pp. 331–349. ISSN: 1751-8814. DOI: 10.1049/sfw2.12030.

[3] Aloisio S. Cairo, Glauco de F. Carneiro, and Miguel P. Monteiro. "The Impact of Code Smells on Software Bugs: A Systematic Literature Review". In: *Information* 9.11 (Nov. 2018). Publisher: MDPI AG, pp. 273–273. ISSN: 2078-2489. DOI: 10.3390/info9110273.

[4] Atlassian. *Bitbucket code review: Merge with confidence*. en. URL: https://bitbucket.org/product/features/code-review (visited on 04/04/2022).

[5] Richard Baker and Ibrahim Habli. "An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software". In: *IEEE Transactions on Software Engineering* 39.6 (June 2013). Conference Name: IEEE Transactions on Software Engineering, pp. 787–805. ISSN: 1939-3520. DOI: 10.1109/TSE.2012.56.

[6] Maik Betka and Stefan Wagner. "Extreme mutation testing in practice: An industrial case study". In: *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. May 2021, pp. 113–116. DOI: 10.1109/AST52587.2021.00021.

[7] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. "Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft". In: *IEEE Transactions on Software Engineering* 43.1 (Jan. 2017). Conference Name: IEEE Transactions on Software Engineering, pp. 56–75. ISSN: 1939-3520. DOI: 10.1109/TSE.2016.2576451.

[8] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. "Identifying the characteristics of vulnerable code changes: an empirical study". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 257–268. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635880. URL: http://doi.org/10.1145/2635868.2635880.

[9] Kaylea Champion, Sejal Khatri, and Benjamin Mako Hill. "Qualities of Quality: A Tertiary Review of Software Quality Measurement Research". In: *arXiv:2107.13687 [cs]* (July 2021). arXiv: 2107.13687. URL: http://arxiv.org/abs/2107.13687.

[10] *Checkstyle*. URL: https://checkstyle.org/ (visited on 08/22/2022).

[11] L. Chen. "Continuous delivery: Huge benefits, but challenges too". English. In: *IEEE Software* 32.2 (2015). Publisher: IEEE Computer Society 50, pp. 50–54. ISSN: 07407459. DOI: 10.1109/MS.2015.27.

[12] Anish Cheriyan, Raju Ramakrishna Gondkar, Thiyagu Gopal, and Suresh Babu S. "Quality Assurance Practices in Continuous Delivery - an implementation in Big Data Domain". In: *2018 IEEE 8th International Advance Computing Conference (IACC)*. ISSN: 2473-3571. Dec. 2018, pp. 7–13. DOI: 10.1109/IADCC.2018.8692131.

[13] S.R. Chidamber and C.F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994). Conference Name: IEEE Transactions on Software Engineering, pp. 476–493. ISSN: 1939-3520. DOI: 10.1109/32.295895.

[14] Istehad Chowdhury and Mohammad Zulkernine. "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities". en. In: *Journal of Systems Architecture*. Special Issue on Security and Dependability Assurance of Software Architectures 57.3 (Mar. 2011), pp. 294–313. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2010.06.003.

[15] *Code Review*. en. URL: https://about.gitlab.com/stages-devops-lifecycle/code-review/ (visited on 04/04/2022).

[16] Juliet M. Corbin and Anselm L. Strauss. *Basics of qualitative research : techniques and procedures for developing grounded theory.* SAGE, 2015. ISBN: 9781412997461.

[17] *Cypress End to End Testing Framework*. URL: https://www.cypress.io/ (visited on 08/22/2022).

[18] N.E. Fenton and N. Ohlsson. "Quantitative analysis of faults and failures in a complex software system". In: *IEEE Transactions on Software Engineering* 26.8 (Aug. 2000). Conference Name: IEEE Transactions on Software Engineering, pp. 797–814. ISSN: 1939-3520. DOI: 10.1109/32.879815.

[19] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. en. 28. printing. The Addison-Wesley object technology series. Boston: Addison-Wesley, 2013. ISBN: 978-0-201-48567-7.

[20] *GitHub features: Intuitive code review tools*. en. URL: https://github.com/features/code-review/ (visited on 04/04/2022).

[21] Gregory A. Hall and John C. Munson. "Software evolution: code delta and code churn". en. In: *Journal of Systems and Software*. Special Issue on Software Maintenance 54.2 (Oct. 2000), pp. 111–118. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(00)00031-5.

[22] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. "Some Code Smells Have a Significant but Small Effect on Faults". en. In: *ACM Transactions on Software Engineering and Methodology* 23.4 (Sept. 2014), pp. 1–39. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/2629648.

[23] Brian Hambling and Pauline Van Goethem. *User acceptance testing : a step-by-step guide*. BCS Learning and Development Ltd, 2013. ISBN: 978-1-78017-167-8.

[24] L. Inozemtseva and R. Holmes. "Coverage is not strongly correlated with test suite effectiveness". English. In: *Proceedings - International Conference on Software Engineering*. Issue: 1 435. IEEE Computer Society, 2014, pp. 435–445. DOI: 10.1145/2568225.2568271.

[25] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017). Conference Name: ISO/IEC/IEEE 24765:2017(E), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[26] Juha Itkonen, Raoul Udd, Casper Lassenius, and Timo Lehtonen. "Perceived Benefits of Adopting Continuous Delivery Practices". In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement ; ISBN 9781450344272* (Jan. 2016). Place: United States, North America Publisher: ACM. ISSN: 978-1-4503-4427-2. DOI: 10.1145/2961111.2962627.

[27] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011). Conference Name: IEEE Transactions on Software Engineering, pp. 649–678. ISSN: 1939-3520. DOI: 10.1109/TSE.2010.62.

[28] *JUnit 5*. URL: https://junit.org/junit5/ (visited on 08/22/2022).

[29] Joseph P. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. "Software complexity measurement". In: *Communications of the ACM* 29.11 (Nov. 1986), pp. 1044–1050. ISSN: 0001-0782. DOI: 10.1145/7538.7540.

[30] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. "Problems, causes and solutions when adopting continuous delivery—A systematic literature review". In: *Information and Software Technology* 82 (Feb. 2017). Publisher: Elsevier B.V., pp. 55–79. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.10.001.

[31] Mika V. Mäntylä, Bram Adams, Foutse Khomh, and Emelie Engström. "On rapid releases and software testing: a case study and a semi-systematic literature review". In: *Empirical Software Engineering* 20.5 (Jan. 2015). Publisher: Springer, pp. 1384–1425. ISSN: 1573-7616. DOI: 10.1007/s10664-014-9338-4.

[32] D. Mårtensson T. Ståhl and J. Bosch. "Test activities in the continuous integration and delivery pipeline". English. In: *Journal of Software: Evolution and Process* 31.4 (2019). Publisher: John Wiley and Sons Ltd. ISSN: 20477481. DOI: 10.1002/smr.2153.

[33] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[34] Rana S. Menshawy, Ahmed H. Yousef, and Ashraf Salem. "Code Smells and Detection Techniques: A Survey". In: *2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. May 2021, pp. 78–83. DOI: 10.1109/MIUCC52538.2021.9447669.

[35] T. Menzies, J.S. Di Stefano, M. Chapman, and K. McGill. "Metrics that matter". In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.* Dec. 2002, pp. 51–57. DOI: 10.1109/SEW.2002.1199449.

[36] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction". In: *Proceedings of the 30th international conference on Software engineering*. ICSE '08. New York, NY, USA: Association for Computing Machinery, May 2008, pp. 181–190. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368114.

[37] *NVivo*. URL: https://www.qsrinternational.com/nvivo-qualitative-data-analysis-software/home (visited on 04/27/2022).

[38] Roy Osherove. *The Art of Unit Testing : With Examples in C, Second Edition.* Manning Publications, 2014. ISBN: 9781617290893.

[39] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. "Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS Project". In: *arXiv:2102.06909 [cs]* (Feb. 2021). arXiv: 2102.06909.

[40] *PMD Source Code Analyzer*. URL: https://pmd.github.io/ (visited on 08/22/2022).

[41] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. "Software fault prediction metrics: A systematic literature review". en. In: *Information and Software Technology* 55.8 (Aug. 2013), pp. 1397–1418. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2013.02.009.

[42] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey". In: *2012 7th International Workshop on Automation of Software Test (AST)*. June 2012, pp. 36–42. DOI: 10.1109/IWAST.2012.6228988.

[43] *ReQtest: Requirements, Test Management, Bug Tracking Tool*. URL: https://reqtest.com/ (visited on 08/22/2022).

[44] *REST-assured | Github*. URL: https://github.com/rest-assured/rest-assured (visited on 08/22/2022).

[45] André Schaffer. "Testing of Microservices". In: *Spotify Engineering* (Jan. 2018). URL: https://engineering.atspotify.com/2018/01/testing-of-microservices/ (visited on 04/07/2022).

[46] Gerald Schermann, Jurgen Cito, Philipp Leitner, and Harald C. Gall. "Towards quality gates in continuous delivery and deployment". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC), Program Comprehension (ICPC), 2016 IEEE 24th International Conference on* (May 2016). Publisher: IEEE, pp. 1–4. ISSN: 978-1-5090-1428-6. DOI: 10.1109/ICPC.2016.7503737.

[47] *Selenium*. URL: https://www.selenium.dev/ (visited on 08/22/2022).

[48] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: (2017). DOI: 10.1109/ACCESS.2017.2685629.

[49] Navneet Singh, Vishtasp Meherhomji, and B. R. Chandavarkar. "Automated versus Manual Approach of Web Application Penetration Testing". In: *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. July 2020, pp. 1–6. DOI: 10.1109/ICCCNT49239.2020.9225385.

[50] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. "Grounded theory in software engineering research: a critical review and guidelines". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 120–131. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884833.

[51] Tony Thistoll, Val Hooper, and David Pauleen. "Acquiring and developing theoretical sensitivity through undertaking a grounded preliminary literature review". In: *Quality & Quantity* 50 (Feb. 2015). DOI: 10.1007/s11135-015-0167-3.