Automated Code Inspection: Investigating Deployment of Continuous Inspection

SEBASTIAN STIERNBORG
Automated Code Inspection: Investigating Deployment of Continuous Inspection

SEBASTIAN STIERNBORG
Abstract

Developing high quality software is a challenging task and there are various techniques and processes proposed to achieve high quality of software. This project examines the process of continuous inspection that automatically reviews source code using modern technology, such as repository management services, continuous integration services and continuous code analysis tools. More specifically, this study examines how the continuous inspection process can be deployed effectively in a software team. The deployment was performed together with an interview-based study in order to get feedback on the integration alternatives of the process. The project resulted in a set of guidelines on how continuous inspection can be effectively integrated and how to establish a process that developers perceive as useful and seamlessly integrating into their workflow. These guidelines help to understand the benefits and drawbacks of integrating the continuous inspection process. Further studies are needed to investigate the integration of the continuous inspection process with different tools and features to fully answer the question how the continuous inspection process can be effectively deployed into software team’s development processes.
Sammanfattning

3.4.2 Controlled Experiment .................................. 30

4 Results ..................................................................... 32
4.1 Pre-study ............................................................. 32
  4.1.1 The Principal Company’s Development Processes ... 32
  4.1.2 Principal Company’s Environment ...................... 34
  4.1.3 Developer’s Experience ................................ 35
  4.1.4 Evaluation of Continuous Code Analysis Tools .... 36
4.2 Technical Integration and Implementation ............ 38
4.3 Analysis of the Deployment ................................. 43
  4.3.1 Controlled Experiment ................................ 49

5 Conclusion .................................................................. 51
5.1 Discussion ........................................................... 51
5.2 Future Work ........................................................ 55

Bibliography .................................................................. 57

A Codacy ...................................................................... 62
  A.1 Technical Overview ........................................... 62
  A.2 Workflow in Codacy ........................................... 64

B Examples of SonarQube Issues ................................. 65
  B.1 Blocker Vulnerability Issue ................................. 66
  B.2 Critical Code Smell Issue ................................. 67
  B.3 Major Bug Issue .............................................. 68
  B.4 Minor Code Smell Issue .................................... 69
  B.5 Info Code Smell Issue ...................................... 70

C SonarQube Plugins Utilised ....................................... 71

D Evaluation of Continuous Code Analysis Tools Tables ........................................ 72

E Interview Guide ..................................................... 75
Chapter 1

Introduction

The chapter is structured as follows: a preface, problem statement, research purpose, objective, research goal and question, delimitations, outline of the thesis and related work.

1.1 Preface

This master thesis project will investigate how code review processes can be improved. The project is executed at the Swedish company Furhat Robotics. In this report, the company will be referred as the principal company. Furhat Robotics develops the world’s first social robotics and conversational system platform.

The topic of the project is relevant to the principal company because they have a fast software development pace and constantly aim to shorten their time-to-market. One of the company’s challenges is to rapidly deliver software while maintaining high code quality.

1.2 Problem Statement

Developing high quality software is a persistent challenge which software engineers have been facing. Software quality assurance is a complex task that requires a lot of efforts [1].
One of the traditional practices for improving software quality is the use of code reviews [2]. A code review is a well-documented practice that was formally introduced by Fagan in the 1970’s to improve software quality [2]. Nowadays, in contrast to the original Fagan’s formal process, it is more common to conduct a more modern and less formal code review process [3]. However, the modern reviews can result in highly subjective quality assessment [4].

Usage of static analysis tools is an additional practise that supports developers by inspecting source code automatically and objectively. Static analysis tools constitute a cost efficient and effective way to find bugs. However, they are not commonly used [5]. Research shows multiple reasons for why static analysis tools are not commonly used. For example, the tools’ output is perceived as neither user-friendly nor intuitive. Moreover, the tools have weak support for collaboration [6]. At the principal company, the situation confirms the findings from the previous research. The high paced software development team does not have the resources to use static analysis tools unless they work seamlessly in their workflow.

A recent practise that aims at improving the process of developing software is the continuous inspection process. The process combines static analysis with concepts from both modern and formal code reviews as well as modern software developing processes such as continuous integration [7][8][9]. Continuous inspection enables objective and automatic code inspections by utilising enabling technologies, for example continuous code analysis tools and modern code review tools.

Research has shown numerous positive results of using continuous inspection but there is limited studies of how to efficiently deploy the process in a software development team [10][11][12]. This thesis, therefore, aims at bridging this gap.

1.3 Purpose

The thesis aims at investigating how a continuous inspection code review process, with support from continuous code analysis tools, can be deployed in a viable way. The subject is relevant to the principal company as well as all software development teams that aim at improving their software quality.

There are numerous continuous inspection tools used by leading software developing companies such as Adobe and Siemens [13][14]. However, to the
best of the author’s knowledge, there is limited research investigating the continuous inspection deployment including tools that continuously inspect and support code reviews with static analysis. Therefore, it would be of value for the software development community to investigate how a continuous inspection code reviewing process can be deployed.

1.4 Research Goals and Question

The goal of this study is:

- To investigate the deployment of a continuous inspection code review process.

A central part in the continuous inspection code review process is the enabling technology, particularly continuous code analysis tools that support the process. This study assumes that software development teams already utilise enabling technologies such as continuous integration solutions and repository managers. Consequently, the main focus of this study is the deployment of a continuous code analysis tool that utilises static analysis to produce and manage code quality feedback and metrics.

Subsequently, sub-goals of the study are:

- Investigate a set of continuous code analysis tools and how they technically can be deployed in a software development environment.

- Investigate the architecture of a deployed system that supports the continuous inspection process. For example, how the feedback from the continuous code analysis tool should be delivered to the developer in an efficient way.

- Investigate the effects on code reviews when a continuous inspection process with supporting continuous code analysis tools have been deployed.

The general research question that the project aims at addressing:

- How can a continuous inspection process be efficiently deployed?
1.5 Delimitations

This thesis project will be limited to:

- Only perform a study at one software developing team.
- Focusing on the continuous code analysis tool in the enabling technology of the continuous inspection code review process, i.e., not studying other enabling technologies, e.g., as continuous integration solutions or repository managers.
- Only investigate the process of integrating the selected continuous code analysis tool, SonarQube.
- Performing the implementation only at the principal company.

1.6 Sustainability, Societal and Ethical Aspects

The project aims at providing knowledge to the software development community on how one can improve the process of developing high quality software. Obviously, one prospect is that the work itself can inform and support the community to use practices that create more sustainable software. For example, by improving the development process, engineers can allocate more time to develop new features. Thereby improving the efficiency of the software for the end user. Another example is that by improving software quality the software consumes less computing power by being more efficient.

Furthermore, our society today heavily rely on software. Improving the process of developing high quality software can aid to ensure that the software we depend on is reliable.

To address ethical issues, the interviews performed during this project have all been anonymized to respect the participants’ integrity. All research relevant data collected during the interviews are validated by the interviewee and presented in the results section of the report.
1.7 Outline

The rest of the thesis is structured as follows: Chapter 2, the background chapter covers theory needed to understand the rest of the thesis. Chapter 3, the approach chapter, introduces the research method used for answering the research question. Chapter 4, the result chapter, presents the results from using the methods stated in the previous chapter. Chapter 5, the conclusion chapter, contains the analysis and discussion of the results presented in the previous chapter.
Chapter 2

Background

This project investigates how a continuous inspection code review process, with support from a continuous code analysis tool, can be deployed in a viable way. The background chapter covers the theoretical concepts needed to understand the work of the thesis. In order for the reader to gain an understanding of code reviews processes, the first section presents both traditional and modern code reviews. Furthermore, Section 2.2 presents a modern software development process and how code reviews can be performed in that process using continuous integration and repository managers. Moreover, static analysis that support developers to review code is presented in Section 2.3. Continuous code analysis tools that enable the contemporary code review model of continuous inspection use static analysis techniques to analyse source code to find issues and measure code quality. To further understand the model that this thesis investigates, the continuous inspection approach is presented in Section 2.4. The continuous code analysis tool, in this study SonarQube, is a central enabling technology for continuous inspection. It is presented in detail in Section 2.4.2.
2.1 Code Review

The thesis will focus on Code Reviews the area within the academic field of software engineering. A challenge all software development projects face is to develop and maintain high quality software. Code reviews is one of the most widely used practices that aims at improving software quality by discovering errors unnoticed in the development process [15][16]. The approach demonstrated to have multiple advantages. In particular, it is a cost-effective defect detection practise because it enables early detection when the cost of solving it is low [17][2].

Peer code reviews are manual review of source code that is executed by a developer other than the author of the source code [15]. There are numerous systematic methods on how to perform peer code review. A traditional well-known code review process is Fagan’s formal process. It is a synchronous process based on structured group reviews done in meetings [2]. In contrast, the modern code review processes are asynchronous and more lightweight [3][18].

2.1.1 Formal Inspection

Formal inspection is a code review practise developed by Michael Fagan at IBM in the 1970’s. The process is highly structured, sequential and heavily reliant on the physical presence of the developers during the inspection meetings to inspect printouts of an author’s code and fill in the required paperwork.

An example of formal code inspection usage based on Fagan’s work is NASA’s formal inspection that became a standard in their software assurance work in 1993 [19]. The usage of formal inspection resulted in increased product quality while maintaining cost effectiveness.

The formal inspection process has 7 stages. Each stage consists of completing forms including Inspection Announcement, Preparation Logs, Inspection Defect List, Detailed Inspection Report, Inspection Checklists and other forms. The process has a well-defined procedure regarding by whom and how the forms should be completed. The formal inspection process is further described in NASA’s guidebook [19].

Even though studies proved multiple benefits of these formal processes, one
problem is that they are laborious and time-consuming, which impedes the universal adoption of the techniques [20].

### 2.1.2 Modern Code Review

Recent research shows that modern code review processes are more lightweight, in contrast to formal inspection processes, and use different reviewing environments to support the process, such as mailing lists or web applications [21]. Modern Code Review is defined as a review that is (1) informal (in contrast to Fagan-style), (2) tool-based, (3) asynchronous and (4) focused on reviewing code changes [3][18].

In contrast to the highly structured and synchronous formal code inspection process that requires the inspectors to sit down in a room to perform an inspection meeting, a new process arose during the late 2000s. Open source projects started to perform code reviews asynchronously via the use of email [18]. Using this procedure developers send source code patches using email lists, while the project members evaluate them and request modifications if needed. When the quality level is deemed to be sufficient, a core developer of the project commits the patch to the codebase. This process is visualised in Figure 2.1. Research has shown that core developers break the practise of submitting reviews before committing patches to the codebase [22].

![Asynchronous code review process from the late 2000s.](image)

Figure 2.1: Asynchronous code review process from the late 2000s.
To improve the informal and unstructured code reviewing processes, modern code reviews use tools such as web based Gerrit or Gitlab. In comparison to the amount of bureaucracy which formal code inspections require, these tools seamlessly distribute the source code and reviews online, i.e., improve both the quality and quantity of reviews [23][24]. Figure 2.2 displays a developer’s view of a web-based code review platform. An example of a modern code review process, the pull-based software development model, is further presented in Section 2.2.2.

The main challenge in code review is the substantial amount of human labour that is required even when a code review environment, such as Gerrit, is used [25]. Moreover, software companies like Google require at least one human review before committing a source code change [18]. Apart from the effort required, research points out that code reviews have a steep learning curve to understand defect patterns and conventions [25][26][17]. Furthermore, in contrast to formal inspections, research shows that, in the informal process, acceptance and response time are related to social factors, for example reviewer load and author’s experience [4].
Figure 2.2: Gitlab’s Merge Request web interface that can be used to review code.
2.2 Software Development Life Cycle

There are numerous methods to develop and maintain software, formally known as Software Development Life Cycle (SDLC) in the industrial fields of engineering and computational sciences [27]. In the late 1990s, new methodologies, Extreme Programming, SCRUM, DSDM and others, gained significant interest [28][29]. The new methodologies accentuated “close collaboration between the development team and business stakeholders; frequent delivery of business value, tight, self-organizing teams; and smart ways to craft, confirm, and deliver code” [29]. In 2001, the term Agile was applied to this group of methodologies.

One of the main concepts of Agile methods are that they are iterative [8]. In traditional sequential methods code is integrated in the end of a project. Closely related to modern agile methods are Continuous Integration (CI), presented in Section 2.2.1. It is an agile practise that focuses on integrating code more frequently during the development process [30]. In addition to continuous integration software developing teams normally use version control systems that can utilize workflows, presented in Section 2.2.2, that support working with the agile approach.

2.2.1 Continuous Integration

In projects using traditional sequential SDLC models, the code integration is considered as a separate phase in the end of the project where all developers integrate their code in the mainline [30]. This practise has the disadvantage that problems may only arise in the end of the project when approaching delivery [30]. Continuous integration is an agile practise in software development aiming at speeding up the delivery of software by decreasing the integration overhead. This is achieved via more frequent integration during the development process [30].

In 1991 Grady Booch proposed the term CI [31]. It later gained popularity in the Agile Extreme Programming (XP) methodology community. In 2006 CI was revised by Martin Fowler. Fowler defines CI as “a software practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.” [32].
The aim is to make integration an “non-event” by integrating continuously during the development process instead of doing it all in the end [30]. This results in a shorter feedback loop that enables developers to fix problems early. Typically, using the continuous integration practice each change to the source code, for example, a checked in commit in a version control system, triggers a new build that automatically runs tests to verify that the code can be integrated. Developers then automatically get feedback if their code pass the tests presented in, for example, an email [32].

Modern continuous integration software provides simple configurations of the whole process from build servers to support for source control systems and custom feedback notifications, examples of tools are Jenkins, Travis CI, Gitlab and CircleCI [33][34][35][36]. An overview of the continuous integration process is presented in Figure 2.3.

![Figure 2.3: Overview of the continuous integration process.](image)

### 2.2.2 VCS Workflow

Version Control Systems (VCSs) are used when developing software to record and control changes to files [37]. One of the most common VCSs today is Git. Using Git, a developer clones the full repository into his/her working copy and develops it independently. A common workflow of developing using Git is to
branch out of or fork the mainline to later integrate the source code by merging back into the mainline [37]. In other words, copy the main source code (mainline) into an independent version (branch or fork) that one can develop and then integrate back into the main source code when the work is done.

![Diagram](https://example.com/diagram.png)

**Figure 2.4:** Pull request workflow using Git on hosted services like Github.

As Git gained popularity, the services for hosting Git repositories arose, for example, GitHub, Bitbucket and GitLab [38][39]. These services offer functions that aim at streamlining the distributed version control system experience. In particular, one function these services offer is a code review interface, named Pull Requests or Merge Requests. This is a graphical user interface that enables developers to create a request to merge source code when a set of changes are ready to be submitted to the mainline. Figure 2.4 shows the overall workflow using pull or merge requests. When a request is created, using the hosted services, a web interface (shown in Figure 2.2) enables the developers to review and discuss the changes before approving or declining the request to merge [38]. Another common tool is Gerrit that integrates with Git to allow a similar workflow that supports the code reviews through a web interface [25].

Using GitLab, the repository can be set up to ensure that a number of reviewers have approved the change before it can be merged [39]. Additionally, GitLab can also be set up to only approve merges when external services have approved the request. For example, a continuous integration server has successfully run through test and validated the code or other services as code coverage reporting and continuous inspection services.
2.3 Static Analysis

Static analysis is a practise that aim to improve software quality by evaluation code without executing it. Static analysis differs from conventional testing that requires design and construction of test cases. Furthermore, it also differs from code reviews because it does not require human labour from anyone except the developer that runs the static analysis tools that automatically generate feedback.

Static analysis tools provide data on how software complies to recommended programming practices [5]. For example, static analysis tools can detect issues by searching for defective code patterns or building a model of the source code and perform a symbolic execution, i.e., traversing the model, to find issues. One of the first commonly used static analysis program was the linter for C programs that flags, for example, unused variables and pointer alignment problems [40].

Another example is the tool FindBugs that investigates Java source code to rapidly find coding defects in categories such as bad practice, performance, malicious code vulnerability [5]. The user specifies all source directories and runs the analysis. Upon completion, the FindBugs tool presents a graphical interface to navigate through the warnings found in projects source code, as seen in Figure 2.6. In the application in-depth information about the warning can be found, for example, what is the potential issues and why it might be a problem.
It has been demonstrated that static analysis tools improve software and are faster and less costly than manual code inspections [41][42][43][44]. However, research shows that many companies do not frequently use them partly due to the fact that they are non-user-friendly and have weak support for usage in teams [6]. Johnson, Song, Murphy-Hill and Bowdidge state in the conclusion of their article: “Future static analysis tools could improve adoption by software developers by enhancing support for team development while using static analysis tools, improving integration of the tool into developers’ processes, having intuitive defect presentation and detailed explanation of defects with automatic fixes where appropriate, and including easy and useful configuration options for the tool.” [6].
2.4 Continuous Inspection

Software quality assurance is a complex task that normally requires a lot of efforts and expertise. Continuous Inspection is a holistic process that aims at simplifying quality assurance making it an integral part of the software development life cycle. It was introduced by both SonarSource and PRQA in 2013 [11] [7]. Certain technicalities in the two definitions of the models differ. However, they both aim at raising the visibility of software’s quality for all stakeholders to be able to improve quality.

PRQA’s states that Continuous Inspection is a modernization of formal code inspections that confronts weaknesses from formal code inspections, peer code reviews and testing. It achieves it mainly by creating well-defined (1) inspection criteria and an (2) inspection process that is agile and continuous with (3) enabling technology to support the stakeholders.

1. The inspection criteria are used to enforce standardised validation at the end of each code inspection. The inspection criteria are divided in “functional requirements that describe the behaviours needed to support the user’s needs and structural requirements that identify the attributes that address the internal integrity of the system” [7]. The functional requirements, or external quality requirements, will typically be verified by software testing. The PRQA article emphasises that the code inspection criteria need to include structural requirements, or internal quality requirements, that verifies the quality and integrity of the software with quantifiable measurements. Traditional inspection criteria in formal inspection require a checklist or code standard that solely depend on the involvement and experience of the reviewers [4]. The main idea of process’ inspection criteria is not only to set a standard that all source code needs to comply with but also to set the criteria that are automatically assessed by enabling technology.

2. The inspection process shall encourage all stakeholders to continually improve both the code and development process. The process includes the principle of “early and often” defect discovery using automated builds and continuous integration to regularly and autonomously perform code inspections with the previously mentioned inspection criteria. PQRA’s article accentuates that it is vital that the inspection process involves all stakeholders and embrace collaboration to achieve a successful continuous inspection process. Furthermore, the inspection process shall
enable analysis of measurements from the automatic inspection criteria assessment. In addition to support for analysing the assessment data, there should exist a complete traceability of all metrics and a process to address the detected issues. All stakeholders should be able to trace and analyse trends in the assessment data. If the code cannot comply with the inspection criteria and needs to be integrated, there should be a well-defined deviation principle, for example, that an unfixable issue needs to be confirmed as a “won’t fix” issue by multiple stakeholders.

3. The enabling technologies is a fundamental part of the Continuous Inspection model that support the inspection process. A more in-depth description of the enabling technologies is presented in Section 2.4.1. In general, enabling technologies include tools that can automatically continuously assess the inspection criteria, collaboration solutions that present the assessment data and solutions to store and aggregate data from the code inspections.

Software quality assurance becomes a central part of the software development life cycle by using continuous inspection paradigm. Consequences of using continuous inspection include:

1. Short feedback loops have double benefits because they allow the developers to both quickly resolves problems early and cost efficiently as well as at the same time educate developers [11][12].

2. Software teams gain ownership over the code and receive continuous feedback on the quality including the evolution of quality [11][12].

3. By using an automated objective quality assessment, the process improves the reviewing quality and efficacy at the same time. Therefore, it helps to create a uniformity in the requirements to integrate source code [10].

4. Overconfidence in the tools resulting in that developers lose focus on code quality [12].

5. Poorly configured code analysis results in an overflow of detected false positive issues. Consequently, it causes the developers to ignore all issues. Furthermore, it is vital that the developers understand the motivation and benefits of continuous inspection. Otherwise, they may perceive that the process disturbs and constrains their creative process [12].
2.4.1 Enabling Technologies that Support Continuous Inspection

Enabling technologies is a central part of the continuous inspection process. Today there are numerous technologies that can be utilised to support the continuous inspection process in different stages. They can be divided into three categories:

1. **Repository managers** that enable collaboration, traceability and documentation. Services like GitHub, Gitlab and Bitbucket allow teams to control both their source code and code reviews as previously mentioned in Section 2.1.2 and 2.2.2.

2. **Continuous integration solutions** that can be integrated with the repository managers to run tests and tools allowing to continuously utilise prompt feedback.

3. **Continuous code analysis tools** that analyse source code, mainly by using static analysis, to perform automatic code inspections to detect bugs, code smells and security vulnerabilities.

Both repository managers and continuous integration are well researched areas. In this study the focus is on continuous code analysis tools which are less explored. The continuous code analysis tools, in addition to performing automatic code inspections, empower collaboration within the team to manage detected issues. They provide a user interface to view the stored and aggregated review data. Every platform has its strengths and weaknesses, they integrate with different services, support different languages, execute the analysis in different ways, etc. Continuous code quality tools are also commonly referred to as continuous inspection tools or code quality platforms. They include CodeBeat, Code Climate Quality, CodeFactor, LGTM, Scrutinizer, SonarQube and Codacy [45][46][47][48][49][50][51]. The simple evaluation of the tools presented in Section 4.1.4 concludes that SonarQube is the most appropriate tool to implement in the principal’s environment.

A typical implementation of a continuous inspection process with the three previously mention technologies is presented in a recent research article [10]. The implementation utilises the following services:

1. The repository manager GitHub was used as a web-based hosting service for version control, code reviews and team collaboration.
2. The continuous integration service Travis CI was used to automatically build and run unit tests to ensure that the functional, or external quality, requirements are satisfied.

3. The continuous code analysis tool SonarQube execute code analysis to ensure that the structural, or internal quality, requirements are satisfied.

The feedback data from both the continuous integration service and the continuous code analysis tool are posted automatically in the repository manager’s pull request view, as seen in Figure 2.2. The continuous code analysis tool, SonarQube, automatically writes comments as a peer review assignee would in the pull request view. The author is then responsible to resolve all comments and failing tests by both these external services as well as a peer reviewer before integrating the changes in the mainline. An overview of the process is presented in Figure 2.7.

![Figure 2.7: Overview of a continuous inspection code reviewing process.](image)

2.4.2 SonarQube

The continuous code analysis tool SonarQube is a web-based open source platform that supports continuous inspection process by performing automatic reviews based on dynamic and static code analysis [50]. The platform detects
bugs, code smells and security vulnerabilities for over 20 programming languages. Furthermore, it collects and aggregates all data, for example, about code coverage, code complexity, unit tests, bugs and duplications. The platform presents a dashboard for each project that summarises the current quality of the project using metrics. An example of a project view and an issue overview is shown in Figures 2.8 and 2.9. In addition, users can effortlessly use the web interface to track metrics over time, read up on potential issues with detailed explanation of the defects and assign them to the developers or mark them as false positives.

In addition to centralising quality control, SonarQube integrates with repository managers, continuous integration tools, and code reviewing platforms. An example how SonarQube can be integrated is presented in Figure 2.7. The workflow of using the platform is addressed next.

Figure 2.8: Example of a project’s dashboard view in the SonarQube’s web interface.
Figure 2.9: Example of a project’s issue view in the SonarQube’s web interface.

**Workflow in SonarQube**

SonarQube’s source code analysis tool, SonarScanner, runs code analysis, for example, on the CI server, as a part of the CI pipeline when a developer checks in code to the repository management service. An overview example of the process using SonarQube is presented in Figure 2.7. The scanner’s raw analysis data is then sent to the SonarQube server’s compute engine which analyses the reports. The engine posts general metrics such as code coverage, duplications and so on into the platform. In addition, issues are created by the engine every time source code does not comply with the rules set up in the project’s quality profile. Each issue is classified as either a bug, vulnerability or code smell. Moreover, a severity label is defined. Examples of issues can be found in appendix B. SonarQube Issue’s severity levels are [52]:

1. **BLOCKER**: "Bug with a high probability to impact the behaviour of the application in production: memory leak, unclosed JDBC connection, ... The code MUST be immediately fixed."
2. **CRITICAL**: "Either a bug with a low probability to impact the behaviour of the application in production or an issue which represents a security flaw: empty catch block, SQL injection, ... The code MUST be immediately reviewed."

3. **MAJOR**: "Quality flaw which can highly impact the developer productivity: uncovered piece of code, duplicated blocks, unused parameters, ..."

4. **MINOR**: "Quality flaw which can slightly impact the developer productivity: lines should not be too long, "switch" statements should have at least 3 cases, ..."

5. **INFO**: "Neither a bug nor a quality flaw, just a finding."

When created, the issues have status *Open*. They are automatically assigned to the source code author of the issue. The author can confirm the issues and change the status to *Confirmed* and directly commit a fix or assign the issue to another developer. If the commit fixes the issue, the platform will automatically mark the issue as *Closed* with a *Fixed* label. Furthermore, users can manually change the issue statuses to *Resolved* by marking them as *False Positive* or *Won’t Fix*.

In addition, for each scanning the platform checks if the results comply with the projects Quality Gate, i.e. *inspection criteria*, and reports a status: *passed* or *failed* [53]. The Quality Gate can be set up by the project administrator to enforce rules, for example that 80 % code coverage is required or that no code smell issues nor bug issues shall be added.

If SonarQube’s branching and repository management service integration, for example Gitlab, is enabled, the results of each CI build is automatically reported. This can be reported by posting the quality gate status as a CI pipeline stage or as SonarQube comments that present the issues detected in the checked-in changes. Top issues are commented with links to the issues code location in the repository managers web interface and a quick access link to the SonarQube’s issue information view. Figure 2.10 shows the SonarQube automatic code review comment in a merge request. Figure 2.11 shows the issue view that presents the developer with further information about the issue. It normally includes code examples of the problem and how one could fix it.

When the developer has been notified in the merge request that feedback is available, he or she reviews the feedback by inspecting the issues. If a code change is needed the developer can check in all the modifications to trigger a
new analysis. It allows the developer to make sure that all issues have been resolved and that no new have been created.

<table>
<thead>
<tr>
<th>Discussion</th>
<th>Commits</th>
<th>Pipelines</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SonarQube Robot started a discussion on commit 997c2091</td>
<td>3 weeks ago</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SonarQube analysis reported 116 issues</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 12 blocker</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 13 critical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 11 major</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 50 minor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top 50 extra issues
Note: The following issues were found on lines that were not modified in the commit. Because these issues can’t be reported as line comments, they are summarized here:
1. Add or update the header of this file
2. Reserved name used for macro (begins with underscore followed by a capital letter)
3. Add or update the header of this file
4. Reserved name used for macro (begins with underscore followed by a capital letter)

Figure 2.10: SonarQube’s commit/merge request comment on Gitlab.

Figure 2.11: SonarQube’s issue view.
Technical Overview

This Section will give a detailed technical overview of SonarQube. The general architecture of the platform is presented in Figure 2.12. Essentially, the SonarQube Platform consists of 4 components [54]:

1. **SonarScanner**. It runs the analysis on the source code, normally from the build or continuous integration server, and reports the results back to the SonarQube Server [55]. This includes both static analysis on source code, compiled code and dynamic analysis for certain languages. The SonarScanner is a Java program that runs the analysis on all supported languages and can be extended by component 4, SonarQube Plugins. During the analysis, the configuration data is requested from the SonarQube Server. Meanwhile, files need to be provided locally at the location of the execution. Finally, the data from the analysis is sent back as a report to the SonarQube Compute Engine Server.

   In certain cases, the process of compiling and generating code analysis data is automated, e.g., for analysing Java projects with build tools such as Maven or Gradle. In other the cases, for example, for analysing C and C++ code, the analysis process depends on the external configurations to generate analysis data.

2. **SonarQube Server** consists of 3 processes [54]:
   
   (a) Web Server that hosts the web application’s user interface of the platform to present the data from the SonarQube Database. Two views from the interface is shown in Figures 2.8 and 2.9.

   (b) Search Server is based on the search engine Elasticsearch that is used for all searches in the platform’s web-interface. The search functions can be used, for example, to find a specific file in a project.

   (c) Compute Engine Server analyses the reports from the SonarScanner and saves the data to the SonarQube Database.

3. **SonarQube Plugins** are extensions that can expand the platform’s features, including language support, version control support, authentication and governance. There are numerous plugins that are officially supported by SonarSource available as well as an extensive selection of community supported plugins. These include plugins allowing further language support, 3D visualization of code metrics, support for authentication services as Google Authenticator and integration with services
as GitLab [56].

There are three technical stacks that a SonarQube plugin can extend[57]:

(a) SonarQube Scanner, allows developers to extend analysis of source code. For example, external static analysis tools or coverage tools reports can be read by the scanner and reported to the Compute Engine. For example, SonarQube’s community C++ plugin uses common tools such as valgrind, cppchecker, etc.

(b) Compute Engine, by consolidating the output of the scanners. For example, by computing second level measurements, assigning new issues, aggregating measures and storing the data to data stores.

(c) Web Application, for example by introducing additional views for visualizing code metrics or integrate authenticating services.

4. **SonarQube Database** is where all the platform’s data is stored. It includes configurations, quality snapshots, views, issues and rule profiles. Supported database engines are PostgreSQL, Microsoft SQL Server, Oracle and MySQL.

Figure 2.12: SonarQube’s platform architecture [54].
Chapter 3

Approach

The project investigates how continuous inspection process can be deployed effectively. The investigation is performed by conducting a case study that includes an implementation and deployment of continuous inspection together with consecutive semi-structured interviews to investigate how the developers perceive the deployment.

Before conducting the implementation and deployment of continuous inspection, the pre-study is conducted. It aims at investigating and analysing the principal’s current code review process with semi-structured interviews. The interviews aim to identify the workflow, technical conditions and opportunities. With the identified environment, a simple evaluation of the continuous code analysis tools that can support the code review process is carried out to select a tool to further investigate and deploy.

The second part of the project is to implement and deploy the selected tool, SonarQube, in the principal’s software development life cycle. To investigate the deployment, the study includes multiple semi-structured interviews during and after the implementation. In combination with the interviews, a controlled experiment is planned to be performed to investigate if the interviews’ subjective results concerning the deployment align with the controlled experiment. Finally, an analysis of the results is performed to define guidelines for the deployment of the continuous inspection process.

Figure 3.1 presents the principal’s code review process from the pre-study, before continuous inspection was deployed. Figure 2.7 from Section 2.4.1 visualises the expected post-implementation code review process, i.e., after the continuous inspection process supported by the continuous code analysis
tool SonarQube has been deployed.

Figure 3.1: Overview of the pre-implementation code reviewing process at the principal company. Based on the results of the pre-study, presented in Section 4.1.1.

3.1 Pre-study

To acquire background information and investigate how the principal’s software development team executes code reviews, an interview pre-study has been performed. The pre-study investigates the principal’s technical conditions, software development life cycle processes and other practices. In addition, the pre-study includes an evaluation of code analysis platforms that support the continuous inspection process and fit the principal company’s environment.

3.1.1 Interview Study

An interview study has been conducted in the company’s software development team to gather the background information. 5 members of the team were interviewed in one-to-one to obtain data. Moreover, the interviews were conducted with semi-structured interviewing techniques. This interviewing
technique was selected to allow the interviews to be more dynamic and allow interviewees to extend the structured topics.

Following this method, an interview guide was created. It was used to guide the conversation during the interviews. Furthermore, the interviewer followed up the interesting topics that arose during the conversations and encouraged the interviewees to talk freely, thus complying with the semi-structured methodology. Moreover, the interview process used recommendations gathered in Hove’s and Anda’s research work on semi-structured interviews [58].

The information collected during the interviews aims at studying how the team conducts code review pre-implementation. At the same time, information shall be gathered regarding the team’s general development processes, environments, systems, tools and languages with the purpose of understanding what solution can fit and how it could be integrated into their current environment.

In addition to the company’s current processes and environment, the information was collected about developers’ own experiences in related areas such as code smells, continuous integration, static analysis, programming language experience and how they previously worked with code reviews. This information was used as a basis for carrying out the analysis of the results of both the analysis study of the implemented tool and the controlled experiment.

In short, three areas were pursued by the interview study:

1. Current Processes – To understand the current software development processes, focusing on the code review process and opportunities to improve it.

2. Technical Environment – To understand how the principal company works with systems, frameworks, tools and which programming languages are used.

3. Knowledge and experience – To obtain data for how to perform and analyse the controlled experiment. In addition, the interview study included collecting information about previous experiences with code review processes.
3.2 Evaluation of Continuous Code Analysis Tools

There are several platforms in the market that can be used in the continuous inspection process. Due to the study’s limited resources, the aim of this part of the study was to carry out a simple evaluation to find a platform that fits the principal company’s development environment. The evaluation is based on the development environment data gathered during the interview study.

Firstly, information about the platforms programming language support was gathered. Thereafter platforms that did not support languages that the development team uses were eliminated as a candidate for the implementation.

Then information was collected about how the remaining tools work and their technical condition. Moreover, an analysis was done to ensure that they are functional in the principal’s development environment. Lastly, an economic evaluation of the platform cost was performed to match the conditions gathered during the interview study.

A weakness is that the study restricts itself to one tool strictly dependent on the conditions at the principal company. The study is simply evaluating that the selected platform is a good fit to the principal’s environment. Since this evaluation is not the main contribution of this project, this weakness has deliberately been allowed as a rational limitation of the study.

3.3 Technical Integration and Implementation

To deploy the continuous inspection workflow only one continuous code analysis tool is integrated and implemented at the principal company in this study. The integration and implementation are carried out with the tool SonarQube that was selected in the evaluation presented in Section 4.1.4.

3.4 Analysis of the Deployment

To investigate the deployment of continuous inspection and the implemented tool, SonarQube, additional semi-structured interviews were performed. A
controlled experiment was then performed to investigate if the subjective results gathered during the interviews align with the controlled experiment. The results of the subjective analysis together with the experiment are then analysed to draw a conclusion and answer the research question.

3.4.1 Interview Study

During the deployment of the integration at the principal, continuous one-to-one semi-structural interviews were conducted to investigate how the team perceived the deployment. This semi-structured interviewing technique was selected to allow the interviews to be dynamic and allow interviewees to extend the structured topics. The continuous interview process aims to evaluate the configuration at place during the interview to allow adaption of the deployment. For every major change in the configuration all team members were interviewed. For minor changes a minimum of three members were interviewed. This allows gathering of data from different configurations. In addition, post-implementation interviews were conducted after the deployment to gather data of the entire deployment process. The interview guide used can be found in Appendix E.

3.4.2 Controlled Experiment

The controlled experiment was set up to investigate how the developers’ perception of the consequences of the deployment align with the actual effect of using continuous inspection with support from a continuous code analysis tool. The controlled experiment was limited to monitoring only the overall effect of the deployment. More specifically, the experiment was limited to investigating whether merge requests differ with and without continuous inspection. For example, the experiment can investigate the actual difference in issues if developers perceive a difference in the number of issues found with and without continuous inspection. Furthermore, the experiment can give an indication of the efficiently by analysing the time difference to detect issues that are detected both pre and post deployment.

The experiment was conducted by setting up a repository on Gitlab with a clone of one of the principal’s repositories. Five source code files that could be seen as stand alone, to simulate an organic merge request, were selected to be reviewed. An example of a file was a sound synthesizer. For each file
a merge request were created, with and without the tool, and assigned to developers according to Table 3.1. The assigning was performed to ensure that reviews with and without the tool were conducted from both a senior and junior developer. Furthermore, to create less biased data, a developer could not review the same request both with and without the tool. Therefore, the developers with the similar experience and contribution activity were mapped to split the experience for each merge request.

All developers were instructed to perform the merge request as they normally would with the only exception of noting down the time they spend on each request. The merge requests without the tool were all completed before the team got introduced to the tool, i.e., pre-implementation, to ensure that the reviews were not affected by working with the tool.

<table>
<thead>
<tr>
<th>Without tool support</th>
<th>MR 1</th>
<th>MR 2</th>
<th>MR 3</th>
<th>MR 4</th>
<th>MR 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>With tool support</th>
<th>MR 1</th>
<th>MR 2</th>
<th>MR 3</th>
<th>MR 4</th>
<th>MR 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.1: Shows the assignment of the conducted merge requests (MR) with the developers’ id numbers in the cells.

The experiment had a set of limitations to comply with the resources of this project. Time was too limited for the author to produce native merge requests in the principal company’s code base and their merge request history data were too limited to use.

One of the principal’s Java projects was selected for this study by the principal. It was deemed unreasonable to use an open source project and set up identical environments for the tool, both due to the limited time and legal aspects regarding software licenses. In addition, open source projects would require substantial effort for the assignee to review because they are unacquainted with the code. The usage of the principal’s project ensured that the developers have a basic understanding of the project and their native development environment.
Chapter 4

Results

4.1 Pre-study

This section presents a summary of the data that was gathered during the interview pre-study performed with the principal company’s software engineering team in October 2018.

4.1.1 The Principal Company’s Development Processes

This section presents a brief overview of principal company’s software team’s development processes. Internally, the software team uses a flexible and highly adaptive interpretation of the agile software development methodology Scrum. Their interpretation of Scrum is less formal because the team is small and multifunctional. At the time of the interview study, they worked on developing the company’s software and in parallel supporting an ongoing product launch. In summary, the overall process is non-formal and flexible.

In association with the agile methodology, the team practices continuous integration using Gitlab’s continuous integration service. Their overall development workflow is presented in Figure 4.2 and 4.1. It is very similar to the workflow from Figure 2.4 in the background section.
The main focus of the pre-study interview was the code review process, which was used when the pre-study was conducted. The process was summarized by the developers as an informal process integrated with their VCS workflow, as illustrated in Figure 4.2. When the inner development is completed, and the code is deemed as done, a merge request is created in Gitlab. Then the author assigns a reviewer, normally the most active developer of that repository. A continuous integration build validation pipeline stage is reported into
the merge request as an indication for both the author and assignee whether
the project builds. In addition, some of the repositories extend the build val-
ification pipeline stage with a test stage. However, the status (success) of the
continuous integration pipelines is not enforced to be able to integrate code,
i.e., developers can integrate changes that break the continuous integration
pipeline.

Since there is no common practice in the team on how the reviews should be
performed, they partially differ. In common, the assignee inspects the changes
using GitLab’s merge request interface. Using the web-interface, shown in
Figure 2.2, two common practices are used by all developers:

1. Inspect that the amount of code changes and that the files that have been
changed or created are related to the merge request’s issue/feature ticket.

2. Look through the code and check the code style to validate that the code
makes sense and is understandable. For example, that there are no nest-
ing of numerous “for” loops and reasonable variable’s names are used.

An exception was that two developers normally would pull down the code to
their local machine to build and inspect the code using the same practices
instead of only viewing the code in the web interface.

There was a common view that team’s code review process was currently lim-
ited to a lightweight sanity check of the code. A critical assumption, the review
assignees rely on, was that the author of the merge request had tested and ver-
ified all the functionality of the code.

4.1.2 Principal Company’s Environment

In this section the principal company’s technical environment is presented.
Their code base uses numerous programming languages the central ones be-
ing:

- Java, using Gradle as build tool
- Kotlin, using Gradle as build tool
- C, using CMake as build tool
- C++, using CMake as build tool
- JavaScript
The team uses Git as their version control system. To manage their Git repositories, they use the repository manager Gitlab Enterprise Edition that is self-hosted in their VPC (Virtual Private Cloud) on Amazon Web Services. All hosted services need to use SSL and run inside their VPC within a restricted security group following the principle of least privilege (PoLP). All software developers’ development machines run Linux, more specifically Ubuntu version 16.04 or 18.04.

Furthermore, as mentioned in Section 4.1.1, The Gitlab platform is used as the company’s code reviewing platform as well as their continuous integration server. All the company’s projects use project specific Docker containers to build and run the projects in the continuous integration server.

Apart from security, one of the main requirement for the integration of a tool and re-shaped code review process is low maintenance and cost efficiency for a growing team. The principal company is rapidly growing and the software engineering team needs to focus on the core product.

### 4.1.3 Developer’s Experience

<table>
<thead>
<tr>
<th>Developer</th>
<th>Java</th>
<th>C &amp; C++</th>
<th>Kotlin</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer 5</td>
<td>Medior</td>
<td>Senior</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Developer 4</td>
<td>Medior</td>
<td>Senior</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Developer 2</td>
<td>Medior</td>
<td>Senior</td>
<td>Starter</td>
<td>Senior</td>
</tr>
<tr>
<td>Developer 3</td>
<td>Junior</td>
<td>Junior</td>
<td>Starter</td>
<td>Junior</td>
</tr>
<tr>
<td>Developer 1</td>
<td>Medior</td>
<td>Junior</td>
<td>Starter</td>
<td>Starter</td>
</tr>
</tbody>
</table>

Table 4.1: Programming language experience in the principal company’s software development team.

In the pre-study’s interview study, data was gathered about principal’s developers’ experiences. Their programming language experience was classified based on the number of years they had professionally been working with the specific programming language. The data is presented in Table 4.1 and is classified by four categories:

- **Senior**: more than 10 years of professional experience.
- **Medior**: from 4 to 10 years of professional experience.
• Junior: from 1 to 3 years of professional experience.
• Starter: less than 1 year of professional experience.

4.1.4 Evaluation of Continuous Code Analysis Tools

This Section presents the results from the evaluation of the Continuous Code Analysis Tools. Firstly, the tools programming language support results are presented. Secondly, the results of a more in-depth analysis of the two tools Codacy and SonarQube are presented. Additional background information for the selected tool SonarQube is presented in Section 2.4.2 and Appendix A for Codacy.

The language support of all selected tools is presented in Table D.2. The programming language required to support the principal’s environment, presented in Section 4.1.2, were Java, C, C++, Kotlin and JavaScript. All candidates except Codacy and SonarQube were ruled out due to insufficient language support.

C and C++ languages are not officially supported by Codacy. The support is community developed and limited to only using the static analysis tool Cppcheck [59]. In comparison, SonarQube’s C and C++ language support is officially supported and uses SonarSource’s own static code analyser. In addition, SonarQube’s SonarOpenCommunity have an active C and C++ language support plugin that uses both own static analysis engine in addition to supporting over 10 popular static and dynamic analysis tools, over 6 test frameworks and 5 code coverage tools [60]. The SonarQube community version was deemed as capable alternative due to that it includes over 3800 rules in comparison to the official plugin that only includes 310 rules. In addition, the community version supported the principal’s testing framework Google Test which the official plugin did not.

Furthermore, Codacy’s Java and Kotlin language support is more limited than the one from SonarQube [60]. Table D.1 in Appendix D shows which static analysis tools SonarQube and Codacy can integrate with by using official and available community supported extensions. For the JavaScript analysis, SonarQube moved away from JSHint and JSLint and developed their own analyser SonarJS. Meanwhile, Codacy uses JSHint. This evaluation did not take into account if there is any difference in the JavaScript tools performance, primarily due to that the principal’s main languages are Java and C/C++. 
Both Codacy and SonarQube can be integrated in the principals developing environment. Referring to Section 2, Codacy provides officially supported Gitlab integration, while SonarQube supports a community developed integration. SonarQube requires the analysis of the source code to be invoked, preferable on the continuous integration server. In addition, using extended analysis with SonarQube, analysis tools can require report generation before running the SonarScanner. In contrast, Codacy hosts both the platform and analysis on the same server. It invokes the analysis by waiting for a trigger from GitLab.

Both tools can be extended, however Codacy extensions are limited to the code analysis tools, while SonarQube’s extensions can extend both code analysis tools and the platform itself. Codacy is deemed as a less complex solution that is heavily dependent on its official support because it is a closed source tool. SonarQube seems to be more complex, both to perform analysis on the source code and to configure. However, the developer has full control of the tool and there is an extensive open-source community with numerous active extensions available, support forums and documentation.

The infrastructure cost for running Codacy Enterprise, the only version to support Gitlab’s self-hosted enterprise edition, was calculated to be more than 20 times more expensive than SonarQube. The infrastructure costs were calculated on Amazons Web Services using the recommended requirements [61][62]. SonarQube Community Edition with official support from SonarSource for JavaScript, Java and Kotlin and community extensions integrate into the principal’s workflow without any licensing. Using the SonarQube Developer Edition adds official support for C and C++ with a yearly cost based on the size of the codebase. Codacy Enterprise is billed per developer per year.

Consulting with the principal company’s software manager to compare the licensing cost of SonarQube Developer Edition and Codacy Enterprise at the principal company, SonarQube was considered to be substantially lower in price. However, the free SonarQube Community Edition was deemed as an appropriate tool as it has official support for the principals main programming languages, excluding C and C++ where the open-source support was seen to be superior for the principal company.

In summary, two tools, SonarQube and Codacy, support the principal’s central programming languages. However, SonarQube comprehensively extends Codacy’s analysis tools in the principal’s main languages. Together with having a substantially lower cost and more available open-source extensions SonarQube
was considered to be superior to all other tools investigated for the principal’s environment.

4.2 Technical Integration and Implementation

The following section presents a brief summary of the implementation and integration of the selected tool, SonarQube, in the company’s environment to support continuous inspection.

The main requirements that followed from the preliminary study was maintainability and security. This final setup was the result from setting up a number of different experimental hosting arrangements before deploying the implementation to the principal’s production environment. The first hosting arrangement was to host the SonarQube server using SystemD on a Linux server. The whole configuration was manually set through command line interfaces and various configuration files. In addition, Apache 2 was configured as a remote proxy to enforce the principal’s security rules. However, after tuning the environment and upgrading versions it became clear that a containerised solution was superior to minimize the maintenance work and centralise all configuration in code.

The second iteration of the experimental hosting arrangement utilised Docker to create a container image. This made it easier to manage the configuration of the SonarQube server and associated plugin. The container configurations are managed in one file, a Dockerfile, and contains all configuration to run the server. That includes setting up all the dependencies and downloading plugins for the platform as well as database information and credentials. A list of the utilized plugins is presented in Appendix C. This iteration created higher maintainability, however the runtime configuration still needed to be manually entered, such as retention policy and network configurations.

Furthermore, a remote proxy was still required to enforce security and automatically handle SSL renewal and URL setup. The third iteration included running the reverse proxy Traefik as a Docker container thus simplifying the deployment and centralising the configuration in code.

The fourth iteration of the hosting arrangement was deployed into the principal’s production environment. In this iteration, to simplify the complete hosting arrangement of SonarQube, a multi-container configuration was set up using Docker Compose. Docker Compose reads a docker-compose file to
set up all the required configuration, including networking, logging, environment variables and recovery mechanism. This enables developers to deploy the hosting of the services by only installing Docker and Docker Compose on a server and then execute a single command to start multi-container tool. The SonarQube server’s multi-container arrangement contains two containers: a SonarQube Server container and a reverse proxy Traefik container. This hosting arrangement was considered to fulfil the conditions of low maintenance and high security.

Figure 4.3 shows an overview of the system’s architecture in the principal’s production VPC (Virtual Private Cloud). The SonarQube’s server multi-container setup is hosted on a virtualized server, with 2 virtual CPUs and 4 GB RAM, in the principal company’s VPC together with the Gitlab server and Gitlab’s CI runner server. Furthermore, the SonarQube platform’s PostgreSQL database is hosted inside their VPC. The database is hosted on an independent machine with a backup retention policy and traffic is strictly limited to the SonarQube server. Moreover, firewall configurations inside the principal’s virtual private cloud was configured to only allow required interaction for the SonarQube hosting. Figure 4.4 shows a technical overview of the developers workflow using SonarQube.

Figure 4.3: An overview of a implemented system to support continuous inspection.
Each of the principal company’s projects was configured to utilize the platform. After experimenting with different configurations, a stable universal one was established. This setup consists of three steps that need to be conducted for every project:

1. **Setup a Docker container image configuration** for the specific project including all the necessary configurations. This container configuration was heavily dependent on the programming language. For all Java and Kotlin projects with existing Docker configuration no additional Docker configuration was needed due to using SonarQube’s Gradle plugin that setups the required environment. For all other projects a universal Docker container image was configured for each programming language to download the required dependencies for running the analysis. It allows projects’ existing Docker images to easily be extended to run code analysis.

2. **Setup execution configuration for the code analysis.** For Java and Kotlin projects, the configuration to run the code analysis included adding the SonarScanner dependency to the Gradle configuration. In addition, a custom program for Java and Kotlin projects was constructed to parse information relating to the principal’s development process, that included version of the project, branch, connection to CI, etc. For C and C++, the configuration was more complex. A program was constructed to automatically run the analysis for all the principal’s C and C++ projects. The program included building the project with debug configuration, running external code analysis tool to generate reports, parsing versioning data, saving compiler output and lastly running the SonarScanner to report all the data to the SonarQube server. In addition, a sonar configuration file needed to be created for each project, specifying source and include location, language, report locations for external analysis tools, etc.

3. **Setup CI invocation for the execution configuration.** To run the analysis in Gitlab’s CI pipeline a SonarQube job was added in all the principal’s projects. The SonarQube CI job invokes the automated program to run the analysis. After the analysis has been completed at the SonarQube server, a CI job status and commit comment is automatically pushed to Gitlab if enabled.
It emerged that SonarQube’s Gitlab plugin could be improved. For example, CI job statuses had no connection to the SonarQube platform for the developer to directly manage their issues. To deliver enhanced support a custom integration was constructed. The first version was a prototype built in Bash utilising the SonarQube’s API to throw an exit status and URL to the analysis in the system output. This was deemed as inefficient since the user needed to go into the CI log to copy a URL to the analysis at the SonarQube platform.

The second prototype, built in Go, extended the functionality of the Bash version. The output from the SonarScanner is piped into the program. The SonarQube report id is parsed as the entry point to fetch data from the SonarQube web API. The data included number of detected issues, quality gate status, tech debt, code coverage, etc. This information was then compiled into and pushed as an external CI stage together with a link to the SonarQube platform that enabled easy access by pressing the CI status icon. This implementation was however not suitable for a production environment since it was dependent on parsing text output from the SonarScanner and did not work with Gradle. Therefore, a forked custom version of the SonarQube’s Gitlab integration
A plugin was developed to enable this extended integration using SonarQube’s internal API.

In addition to the custom Gitlab integration, a custom CI Dashboard was developed and deployed. This dashboard was constructed as a JavaScript application running on Docker container on a computer connected to a TV screen in an engineering office. The application utilised Gitlab’s API to receive projects and pipeline statuses. An overview of the CI dashboard is shown in Figure 4.5.

To conclude the technical integration, it was straightforward to get the platform set up to analyse a Gradle project from the command line without integrating any support for CI, branches, versions, etc. However, it required substantial effort to construct an automated process that was suitable to run in the principal company’s production environment. The technical work included:

- Deploying infrastructure.
- Ensuring security.
- Configuring a container setup for hosting.
- Automating analysis.
- Developing a standardised and efficient container configuration to run the code analysis.
• Developing programs to be able to automatically run the analysis adapted to the principal’s CI build and versioning setup.

• Customisation of the SonarQube Gitlab integration

• Developing a customized CI dashboard.

• Etc.

The evaluation of the tools indicated that SonarQube is a complex tool that does not perform the analysis “out-of-the-box”. The project has demonstrated that indeed SonarQube required extensive work to make it function in a desirable way, especially since this implementation uses the open source plugin for analyzing C and C++. However, this work was considered as tolerable cost to be able to make it work in the principal’s environment.

4.3 Analysis of the Deployment

After the platform had been technically integrated in the principal company’s development environment, a workshop was conducted presenting the Continuous Inspection model and the continuous code analysis tool SonarQube. The workshop included a team discussion on how to establish the inspection criteria and deviation principle. The team decided that the inspection criteria’s functional requirement will be assessed with the current test suite. In the cases where the test suite is incomplete, it was decided that it is the code’s authors responsibility to assess the functional requirements before opening a merge request. Moreover, the inspection criteria’s structural requirements will be assessed by the SonarQube platform. SonarQube’s Quality Gate was set by the team to not allow any issue additions and not more than 5% code duplication. A deviation principle was mutually decided: if the inspection criteria need to be breached for an integration, the project owner must approve the integration. After the functional testing and SonarQube’s inspection, the merge request will be assessed by an assignee similar to the principal’s pre-implementation workflow, as seen in Figure 4.1.

The initial rule profile in SonarQube, i.e., the structural inspection criteria, included all the available rules for each language. After the initial workshop, a meeting was performed with the main stakeholders for each project to skim through the most common rules to adjust the profile and disable the rules that were agreed to be classified as non-applicable. The adaption of the rule pro-
files became an active process during the study with two different practises. For Java, there was one developer, the main stakeholder of the Java projects, who was assigned to manage the rules and actively adjust the profile. For C there was no one responsible for the rule profile management. All developers were encouraged to deactivate and adjust rules as they developed. Without a person to lead the adoption of the rule profile, fewer adjustments were made, and issues became ignored instead of adjusting the profile. The developers stated that they felt that they did not want to adjust the rules by themselves. The projects using a rule profile leader adopted the deployment quicker and developers perceived that it was easier to quickly discuss rules with the leader and disable them if that was mutually decided.

After experimenting with different configurations and workflows it emerged that the planned workflow, shown in Figure 2.7, was not ideal for the developers. The first iteration of the configuration enabled automatic comments from SonarQube. However, the automatic comments in the merge request were perceived as overwhelming in Gitlab. Inline commit comments, shown in Figure 4.6, got posted for every detected issue and global comments, see Figure 4.7, were posted in the merge request at every code check-in. Consequently, this led to a lot of comments and developers’ mail inboxes could be flooded with notification emails. This especially occurred when developers were editing legacy code where SonarQube reported issues as comments for all issues in a source file when only single lines were edited in that specific file. In addition, the comments only gave the number of issues introduced and the issue title and reference in code. To understand the issue and manage it, the developers still needed to use SonarQube’s web interface. A consequence became that the developers ignored the comments and integrated code ignoring SonarQube’s inspection results.
Figure 4.6: Example of multiple global SonarQube comments in Gitlab, which developers felt took focus from human reviewer’s comments.
Figure 4.7: Example of inline comments from SonarQube in Gitlab, which developers perceived as inefficient partly due to that the reference linked to the SonarQube rule not the detected issue.

Another iteration disabled automatic comments and instead relied on that the developer checked the platform before integrating code. This was quickly deemed as a unviable solution because developers perceived it taking a lot of time to manually check the status of the branch before merging. Consequently, developers merged code with unmanaged issues.

After experimenting with altering configurations for different projects the most efficient workflow emerged. This configuration disabled automatic commenting and instead only posted SonarQube’s Quality Gate status as a CI pipeline job at every code check-in. This allowed the developers to promptly and simply see the gate status in a way they perceived as “more natural to the development process”. An example of a CI pipeline is shown in Figure 4.8. However, they found it annoying that the CI Job did not have a link to the SonarQube platform. Support for this was developed in a custom version of the Gitlab plugin presented in the previous Section that was planned to be deployed after the version had undergone testing.
Figure 4.8: Example of a CI pipeline that include SonarQube as an external CI Stage

The usage of the CI pipeline stage changed the workflow to fix issues in a more incremental way, in the inner development process, before creating a merge request. Developers perceived that the feedback became less intrusive. However, when the quality gate status was available as a CI pipeline stage, it was strictly enforced by Gitlab to pass all CI stages to be able to merge the code into the mainline from a merge request. An example is shown in Figure 4.9.

Figure 4.9: Merge request with failing pipeline cannot be merged into the target branch.

The engineering managers stated they would like to have a better overview of the projects and the quality status even though the SonarQube platform has a project dashboard. To facilitate this feedback a custom CI Dashboard was developed and deployed in the engineering office. A feature that played a siren sound if mainline had a failing pipeline had a clear effect. It assured that no developers pushed to the mainline without assuring that the changes had successfully passed a CI pipeline.

The resulting development workflow is presented in Figure 4.11. The post-implementation continuous inspection code reviewing process is presented in Figure 4.10. It differ from the expected process presented in Figure 2.7.
Another significant finding from the deployment was the importance of the CI runtime. Developers did find it critical to have quick feedback from SonarQube’s CI stage. Therefore, adaption of the principal’s slower CI Pipelines were made by utilising more computing power and implementing more efficient practices for example caching. In addition, developers found it to be efficient to separate CI stages to isolate each type of test, for example integration test, unit test, code inspection, etc. This allowed the developers to visibly get feedback on what type of stage fails and more importantly the separated stages enabled quicker feedback due to that the splitting allowed CI jobs to be parallelized. This resulted in a significant reduction in CI runtimes.

In general, the development team found the tools to be intuitive and easy to start using during the deployment. There was a strong perception from the whole team that the continuous inspection process improved the code quality. In addition to the new code review process the team appreciated an overview and issue management function of the SonarQube platform.

Figure 4.10: Overview of the actual post-implementation code reviewing process at the principal company.
4.3.1 Controlled Experiment

After a continuous inspection process with SonarQube has been implemented it became evident that the expected workflow was not aligned with the workflow developers deemed as most efficient. Consequently, the controlled experiment that, for example, could investigate the difference in detected issues, became inoperable.

The expected workflow, visualised in Figure 2.7, was assumed to be highly similar to the pre-implementation workflow, seen in Figure 4.1. It assumed that the user would receive the feedback from the SonarQube platform through automatic comments in the repository manager’s merge request view. The assumption, therefore, included that feedback would only be reviewed when the development was deemed finished and ready to merge. Instead, in the deployed process developers preferred to check SonarQube’s Quality Gate status as a part of the CI pipeline after every code check-in. It became clear that disabling comments and instead showing the Quality Gate at every code check-in was more efficient to encourage developers to use the SonarQube platform and incrementally fix issues.

The post-implementation part of the controlled experiment could not be per-
formed in the expected way. Developers used the CI stage status together with SonarQube’s web view continuously during the development and did not utilise the merge request view SonarQube comments. Consequently, it became inoperable to perform an experiment that was based on automatic comments that were disabled in the actual deployment. Future work shall adapt the controlled experiment to the deployed process. Unfortunately, this study’s limited resources do not allow that adaption.
Chapter 5

Conclusion

5.1 Discussion

This section starts with discussing the technical integration work and then presents a number of deployment guidelines to answer the research question: how a continuous inspection process can be deployed efficiently.

One of the main requirements the principal stated in the pre-study was that the solution that is integrated into the company’s production environment would require low maintenance. After testing several hosting arrangements, it became apparent that a container-based hosting arrangement created a maintainable solution. By using the Docker and orchestration program Docker Compose, all configuration including dependencies, network settings, retention policy’s, etc. is written down into single configuration file and can easily be deployed with a simple command. This hosting arrangement makes it easy, for example, to migrate the services, upgrade them, scale them up etc.

Another main requirement was security. The principal company required that all services use SSL and only have necessary network access. A reverse proxy was therefore set up on the SonarQube server to force SSL and utilise the company’s proxy configuration. To further restrict access, a security group was created in the company’s virtual private cloud that allowed only the compulsory traffic. The SonarQube platform itself was not investigated from a security perspective in this work as it was considered sufficient to limit all traffic to only allow the principal’s internal traffic. If security problems arise, these are limited to the principal’s internal network.
To analyse Gradle projects, the SonarQube Scanner setup was simple. However, to analyse C and C++ projects with the SonarQube’s community C-family plugin extensive work was required. A specific Docker container was developed and required to execute analyses. This container included the SonarScanner, external analysis tools and a program that provided all external analysis tools with configurations to produce reports to SonarQube. This setup allows usage of multiple external analysis tools thus enabling thousands of inspection rules in comparison to the official support that contains 328 rules. However, the setup requires extensive work to automate the analysis to run in a production environment.

To further support the continuous inspection process, custom solutions were developed, for example, a CI Dashboard that visualises the quality information on a large display in the office. In addition, the initial integration between Gitlab and SonarQube was not optimal according to the interview study and required customisations. The integration plugin is a community-developed open-source integration that was non-trivial to setup. There were multiple issues in the integration process and the documentation was limited. However, due to that the project was open-source, an enhanced integration could be developed which, among other things, allowed easier access for the users between Gitlab and SonarQube.

In conclusion, the technical integration required substantial work. The hosting arrangement was considered maintainable and secure with the help of containerisation and traffic restriction. The remaining part of the technical integration required extensive customisation to achieve a solution that the principal considered as operational and highly effective. In the end, the technical integration became sizable and complex with custom developed solutions to make the enabling technology comply with principal’s continuous inspection process. All the developed customizations were built to achieve minimal maintenance. However, maintenance work may be required to support complex customisations. Consequently, a risk of maintenance implies a risk of maintenance costs, excluding the upfront development cost, which questions the cost-efficacy of using free open-source software that requires a lot of customizations.

As a result of the project, it became clear that an efficient integration of continuous inspection process requires not only technical deployment work but also involving the developers in tuning inspection criteria and the format and process of the feedback. The following general recommendations can be defined to facilitate this:
• Conduct a workshop introducing the Continuous Inspection process and the continuous code analysis tool. Present the motivation and benefits of using continuous inspections and propose and explain the workflow. Discuss and assign an inspection criteria leader for every project, for example, the main stakeholder or main contributor. This introduction workshop aimed at resolving the motivation problem mentioned in Merson’s research [12] and was highly valued by the developers in the case study deployment.

• Run initial analysis for each project with all inspection criteria rules enabled. Thereafter, the inspection criteria leader conduct project specific team meetings to assess the detected issue’s rules in each project and set up initial structural inspection criteria. To continuously adapt the profile all project stakeholders shall be able to adjust the criteria by contacting the inspection criteria leader.

• Ensure short CI pipeline stages runtimes. Developers did find it critical to have quick feedback from the CI stages to utilise the continuous code inspection. Recommended practises, that were applied in this study, were for example caching, job parallelisation and complementary computing power.

• Split CI pipeline into detached stages. In addition to short CI stage runtimes, developers found it to be efficient to separate CI stages to isolate each type of test, for example integration test, unit test, code inspection, etc. This allows the developers to visibly get feedback on what type of stage fails. Additionally, separated CI stages contribute to quicker CI stage runtime by enabling parallelization and fast pipeline failure feedback.

• Ensure simple and enforceable feedback in the repository manager. Report only the inspection criteria, named Quality Gate in SonarQube, as a CI stage status and enforce a successful CI pipeline to integrate code. This configuration integrates as a part of the inner development process and encourages the developers to continuously utilise the continuous inspection platform. In comparison, developers found that feedback became overwhelming and hard to comprehend with automated feedback comments from SonarQube in the repository manager. This made the developers ignore the comments and code became integrated without managing detected issues.

• Setup documentation with templates on how to deploy the continuous
code analysis tools to new projects. This allowed developers to easily add the tools to new projects and gain an understanding on how the tool functions. In this study, language specific deployment documentation and templates were created. Templates included a CI configuration file together with a standalone container that extends projects container to automatically run external tools to analyse source code and report the results to the continuous code analysis tool.

- Setup a CI Dashboard that include quality information to further visualize code quality status for all stakeholders. Developers perceived that when there was a display that always presented the project’s quality status, there was more focus on developing the development process and improving code quality. In larger cooperation, one could benefit from having both overview visualisations in shared spaces and team-specific screens in the team’s offices.

These guidelines aim at identifying the issues affecting an efficient deployment of continuous inspection supported by a continuous code analysis tool in a modern development environment. In this study, the modern development environment is limited to the principal company environment. Hence, to make more general conclusions, this work should be extended with additional studies in different companies that have other constraints and environments to investigate deployment of continuous inspection with other enabling technologies. For example, such technologies could be enterprise versions of SonarQube or other tools such as Codacy that seem to have a simpler architecture to execute analyses continuously. Furthermore, using SonarQube together with GitHub that is officially supported by SonarQube may yield different result and require less technical work to integrate.

The essential factor in the deployment process is that the developers have a production process that they use and perceive as proficient, hence their perception of the process is vital. The resulting guidelines of the study are based on developers’ perception of the deployment together with the observations gathered during the technical integration and deployment. For example, the study does not confirm that the deployed process affects code quality or time to conduct code reviews. After the case study’s implementation was completed, it became evident in the deployment that the expected workflow was not consistent with the flow that the developers preferred and found useful. Hence, the controlled experiment became unfeasible because it was designed around the assumption that merge requests would be central in the inspection process. Consequently, there was no time to redesign the controlled experi-
ment. A weakness of this study is subjectivity of developers’ perception of the efficiency of the deployed continuous inspection process that is not validated by a controlled experiment.

An evaluation of the efficiency of the deployment process with respect to the quantifiable impact of the code quality improvement process is not included in this work. Therefore, the conclusion does not fully answer the research question. However, it discusses the technical work that may be required to integrate continuous inspection in a modern software development environment. Additionally, the conclusion gives general recommendation guidelines on how to deploy a continuous inspection process that developers perceive as useful and feasible in their workflow.

5.2 Future Work

The suggestions of future work include conducting a rigorous experiment, deploying continuous inspection in multiple development teams from different industries using various enabling technologies and quantifying the effects of different deployments strategies. Such structures can address some of the limitations of this work and investigate the effects and efficiency of the deployed process with respect to the impact on code quality. To refine this case study an adapted controlled experiment could be performed to further validate the suggested guidelines.

The enabling tools for continuous inspection can also be improved. For example, continuous code analysis platforms can be integrate with IDE’s (Integrated Development Environment) for faster feedback [63]. Future work should investigate the refined technology that can affect the deployment of continuous inspection.

In his article [6], Johnson wrote: “Future static analysis tools could improve adoption by software developers by enhancing support for team development while using static analysis tools, improving integration of the tool into developers’ processes, having intuitive defect presentation and detailed explanation of defects with automatic fixes where appropriate, and including easy and useful configuration options for the tool.” This work using continuous inspection adopts all of the mentioned enhancements except automated fixing. Future work could also include investigating continuous inspection processes with automatic fixes of defects and the usage of machine learning to improve the
process as proposed in a recent research article [1].
Bibliography


[34] Travis CI. 2018. URL: https://travis-ci.org/.


[56] SonarQube - Plugin Library. Accessed on 06.11.2018. 2018. URL: https://docs.sonarqube.org/display/PLUG.


Appendix A

Codacy

Codacy is a web-based closed source platform that supports continuous inspection process by performing automatic reviews based on static code analysis [13]. Information about how the platform works is sparse due to the fact that the platform is closed source, i.e., the source code is not published and there is no documentation on how the platform work. This Section will be limited by the information presented in Codacy’s documentation.

A.1 Technical Overview

Codacy runs all code analysis within the platform. In contrast to SonarQube, no external machine is needed to run the analysis. The platform uses a system of plugin engines that are used to analyse source code. These plugin engines run tools that report potential issues to the Codacy platform [59]. The engines are heavily reliant on external static code analysis tools. All of Codacy’s 42 plugin engines use external open source static analysis tools except one engine for analysing Scala source code that uses Codacy’s Scalameta Pro analyser [59].
Even though the platform is closed source, Codacy allows any developer to integrate other tools by creating own plugin engines and submitting them to Codacy for review. A plugin engine is developed by creating a Docker container that runs the selected analysis tool. The platform provides all source files to analyse and a configuration file containing references to the files to analyse. These files are mounted on the Docker container. The developer needs to write code to invoke the selected tool to analyse the input source files according to the given configuration. Moreover, the developer also needs to print out the results of the analysis in Codacys Result format on standard output (example is given in Listing A.1). The Codacy platform then reads the output of the engine and stores the results. An overview of how the Plugin engine works is presented in Figure A.1.
A.2 Workflow in Codacy

The overall workflow using Codacy is very similar to the workflow using SonarQube. The developer triggers the Codacy platform to run code analysis when checking in code. When completed, Codacy notifies the developer that feedback is ready. In addition, the platform reports the status of the change as a CI pipeline stage and posts a code review comment in the repository management service. The developer can then check the review by reviewing the comment or use the platforms web interface to manage issues. If a code change is required, the developer can check in all the modifications to trigger a new analysis to make sure that all issues have been resolved and that no new have been created.
Appendix B

Examples of SonarQube Issues

B.1 Blocker Vulnerability Issue

Databases should be password-protected

```
Connection conn = DriverManager.getConnection("jdb:derby://memory:myDB;create=true", "AppLogin", "");
Connection conn2 = DriverManager.getConnection("jdbc:derby://memory:myDB;create=true?user=un&password=");
```

Noncompliant Code Example

```
DriverManager.getConnection("jdbc:derby://memory:myDB;create=true?user=un&password=password");
```

Compliant Solution

```
DriverManager.getConnection("jdbc:mysql://localhost/test");
```

String url = "jdbc:postgresql://localhost/test";
props = new Properties();
props.setProperty("user", "fred");
props.setProperty("password", "secret");
DriverManager.getConnection(url, props);
```

Figure B.1: A vulnerability issue in SonarQube classified with the Blocker severity level.
B.2 Critical Code Smell Issue

"indexOf" checks should not be for positive numbers  

| Code Smell | Critical | suspicious | Available Since: October 13, 2018 | SonarAnalyzer (Java) | Constant Issue: 2min |

Most checks against an `indexOf` value compare it with -1 because 0 is a valid index. Any checks which look for values >0 ignore the first element, which is likely a bug. If the intent is merely to check inclusion of a value in a `String` or a `List`, consider using the `contains` method instead.

This rule raises an issue when an `indexOf` value retrieved either from a `String` or a `List` is tested against >0.

Noncompliant Code Example

```java
String color = "blue";
String name = "ilshneel";

List<String> strings = new ArrayList<String> ();
strings.add(color);
strings.add(name);

if (strings.indexOf(color) > 0) { // Noncompliant
    // ...
}
if (name.indexOf("ish") > 0) { // Noncompliant
    // ...
}
if (name.indexOf("ae") > 0) { // Noncompliant
    // ...
}
```

Compliant Solution

```java
String color = "blue";
String name = "ilshneel";

List<String> strings = new ArrayList<String> ();
strings.add(color);
strings.add(name);

if (strings.indexOf(color) >= -1) {
    // ...
}
if (name.indexOf("ish") == 0) {
    // ...
}
if (name.contains("ae") { 
    // ...
}
```

Figure B.2: A code smell issue in SonarQube classified with the Critical severity level.
B.3 Major Bug Issue

"BigDecimal(double)" should not be used

Because of floating point imprecision, you're unlikely to get the value you expect from the `BigDecimal(double)` constructor.

From the JavaDocs:

The results of this constructor can be somewhat unpredictable. One might assume that writing `new BigDecimal(0.1)` in Java creates a `BigDecimal` which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257872792035396256247093725216160642571763. This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed in to the constructor is not exactly equal to 0.1, appearances notwithstanding.

Instead, you should use `BigDecimal.valueOf()`, which uses a string under the covers to eliminate floating point rounding errors, or the constructor that takes a `String` argument.

Noncompliant Code Example

```java
double d = 1.1;
BigDecimal bd1 = new BigDecimal(d);  // Noncompliant; see comment above
BigDecimal bd2 = new BigDecimal(1.1); // Noncompliant; same result
```

Compliant Solution

```java
double d = 1.1;
BigDecimal bd1 = BigDecimal.valueOf(d);
BigDecimal bd2 = new BigDecimal("1.1"); // using String constructor will result in precise value
```

See

* CERT, NUM16-J - Do not construct BigDecimal objects from floating-point literals

Figure B.3: A bug issue in SonarQube classified with the Major severity level.
B.4 Minor Code Smell Issue

"@Deprecated" code should not be used

A code smell issue in SonarQube classified with the Minor severity level.

Figure B.4: A code smell issue in SonarQube classified with the Minor severity level.

Noncompliant Code Example

```java
//
* @deprecated As of release 1.3, replaced by \{@link \#Foo\}
*\/
@Deprecated
public class Foo { ... }

public class Foo {
    /**
     * @deprecated As of release 1.7, replaced by \{@link \#doTheThingBetter()\}
     */
    @Deprecated
    public void doTheThing() { ... }

    public void doTheThingBetter() { ... }
}

public class Bar extends Foo {
    public void doTheThing() { ... } // Noncompliant; don't override a deprecated method or explicitly mark
}

public class Bar extends Foo { // Noncompliant: Fun is deprecated
    public void myMethod() {
        Foo foo = new Foo(); // okay; the class isn't deprecated
        foo.doTheThing(); // Noncompliant; doTheThing method is deprecated
    }
}

See
* [MITRE, CWE-477 - Use of Obsolete Functions](https://cwe.mitre.org/cwe-477.html)
* [CERT, METRO-2-J - Do not use deprecated or obsolete classes or methods](https://www.cert.org/cwe-top-25-controls-java.html)
```
B.5  Info Code Smell Issue

Track uses of "TODO" tags

- Code Smell
- Info
- cwe
- Available Since October 15, 2018
- SonarAnalyzer (Java)

TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later.

Sometimes the developer will not have the time or will simply forget to get back to that tag.

This rule is meant to track those tags and to ensure that they do not go unnoticed.

Noncompliant Code Example

```java
void doSomething() {
    // TODO
}
```

See

- MITRE, CWE-546 - Suspicious Comment

Figure B.5: A code smell issue in SonarQube classified with the Info severity level.
Appendix C

SonarQube Plugins Utilised

1. Branch Plugin for SonarQube Community Edition
2. SonarQube C/C++ Plugin (Community)
3. SonarQube Spotbugs Plugin
4. Sonar GitLab Plugin
5. SonarJava Plugin
6. SonarJS Plugin
7. SonarKotlin Plugin
8. SonarPHP plugin
9. SonarPMD Plugin
10. SonarQube Git Plugin
Appendix D

Evaluation of Continuous Code Analysis Tools Tables
### Table D.1: Codacy’s and SonarQube’s supported code analysis tools [52].

<table>
<thead>
<tr>
<th></th>
<th>Codacy</th>
<th>SonarQube</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Java</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SonarJava (Official SQ)</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td>PMD</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Findbugs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find-sec-bugs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spotbugs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checkstyle</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C/C++</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPPCheck</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>GCC/G++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visual Studio</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visual Studio Core Guideline Checker</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td>Clang Static Analyser</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td>Clang Tidy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC-Lint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RATS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valgrind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vera++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dr. Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SquidSensor (Community SQ)</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td>SonarCFamily (Official SQ)</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td><strong>JavaScript</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SonarJS</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>JSHint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESLint</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kotlin</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SonarKotlin</td>
<td>☑️</td>
<td></td>
</tr>
<tr>
<td>Detekt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table D.2: Continuous Code Analysis Tools Program's programming language support. Green cell implies programming language support maintained and developed by the tool's development team. Yellow cell implies programming language support maintained and developed by the open source community. Red cell implies no programming language support.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Python</th>
<th>COBOL</th>
<th>Swift</th>
<th>Ruby</th>
<th>PHP</th>
<th>Scala</th>
<th>Go</th>
<th>C#</th>
<th>JavaScript</th>
<th>Kotlin</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGTM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeFactor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SonarQube</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scrutinizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeBeam</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeClimate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeHawk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeSlicer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeClimatic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOOLS TABLES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix E

Interview Guide

Potential interview question during the deployment.

- How do you think the platform work at the moment?
- Do you feel that it is easy to use SonarQube?
- Is there anything you believe can be improved?
- Do you feel that this continuous inspection workflow can improve your software quality?
- Do you feel that the tool is disruptive? If so, how do you have an idea on how it could be less disruptive?
- How do you manage issues?
- How does your workflow look? Could it be improved to work more seamlessly?

Potential interview question after the deployment in the end of this research project.

- Did you feel that continuous inspection improved the quality of the software?
- Were the tools easy to start using?
- What were the effects of using the tool?
- Is the code review more than a sanity check now?
- Did the code review process change a lot?
• Did you find more issues using the tool? Bugs, style, vulnerabilities, code smell?
• Did you find it to take less effort to do code reviews on other’s merge requests?
• How were the warnings in the beginning?
• What could make it better?
• How did it compare to your expectations?