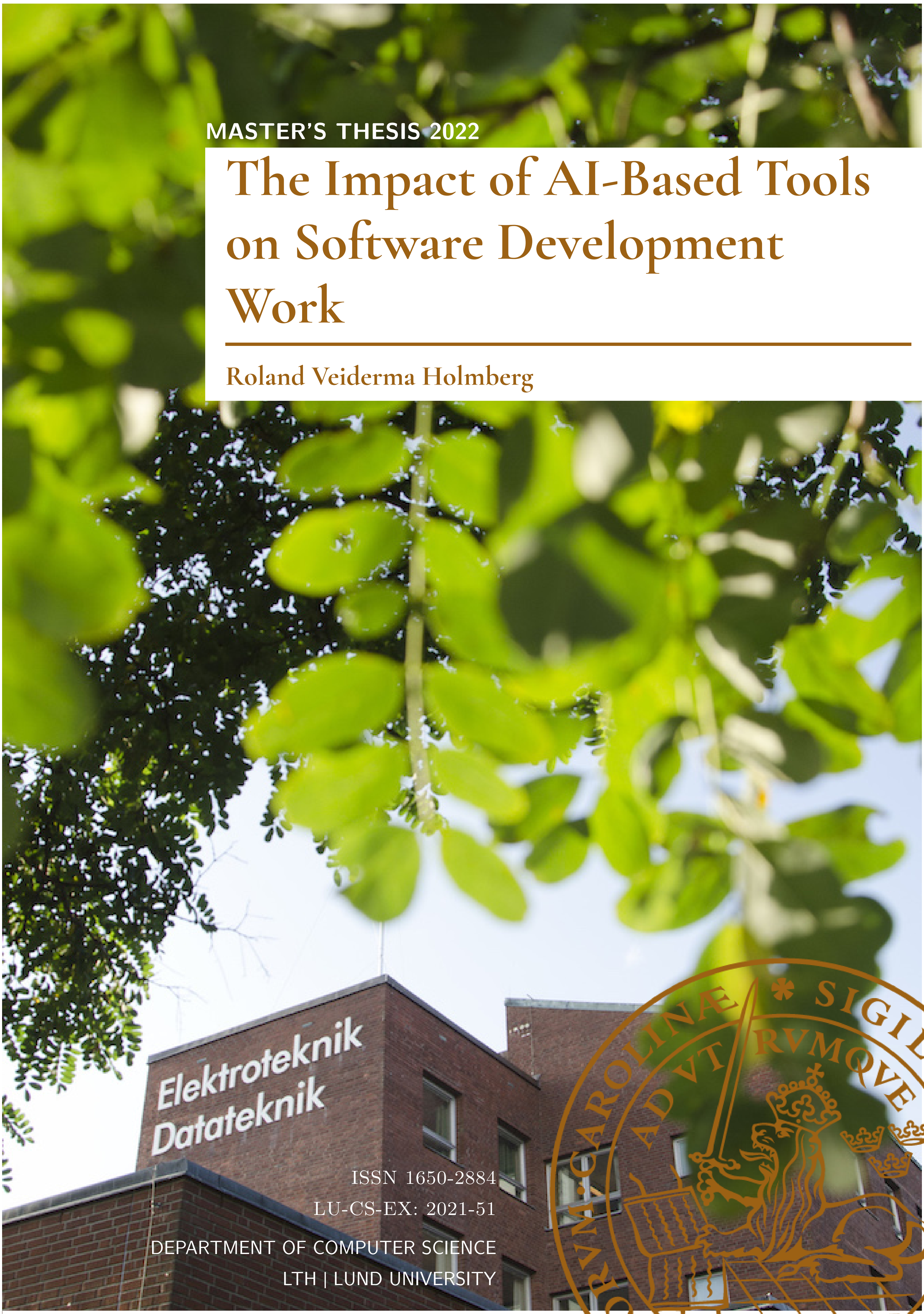
MASTER'S THESIS 2022

# The Impact of AI-Based Tools on Software Development Work

Roland Veiderma Holmberg



ISSN 1650-2884
LU-CS-EX: 2021-51

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-51

# The Impact of AI-Based Tools on Software Development Work

AI-baserade verktygs påverkan på programvaruutveckling

**Roland Veiderma Holmberg**

# The Impact of AI-Based Tools on Software Development Work

Roland Veiderma Holmberg
`ro6142ho-s@student.lu.se`

December 13, 2021

Master's thesis work carried out at Softhouse Consulting Öresund AB.

Supervisors: Björn Granvik, `bjorn.granvik@softhouse.se`
Elizabeth Bjarnason, `elizabeth.bjarnason@cs.lth.se`

Examiner: Per Runeson, `per.runeson@cs.lth.se`

## Abstract

There has been a growing interest recently in using artificial intelligence within software development, even though the idea has existed for more than 20 years. There is a lack of research on software development tools based on machine learning including what tools there are and how they affect the developer. We investigated this by performing a literature review of existing tools, produced a map of these tools and studied thirteen developers using one to three of these tools at a case company. Our findings from the map suggest that the area is under heavy development but still that several tools are mature enough to be used professionally. Our case study findings suggest that machine learning based development tools are less understood and perceived to be less trustworthy than corresponding conventional tools. These are challenges we think are crucial to overcome in order to successfully introduce such tools in professional software development. Our findings can help future research on how such tools can improve and our map can act as a basis on what tools there are available today for future research.

**Keywords**: work environment, machine learning, software development

# Acknowledgements

I would like to thank my supervisors Björn Granvik and Elizabeth Bjarnason for help and guidance in all major parts of this thesis, without their help this thesis would not have been possible. I would also like to thank my examiner Per Runeson for his helpful feedback and guidance during the major milestones of the thesis. For help and guidance in the beginning of the thesis I would like to thank Dag Wahlberg. I would also like to give thanks to all participants of the case study as well as all of the project managers whom arranged it possible to perform the case study. I would like to thank Ola Persson, and also Björn Granvik once again, for finding projects to perform the case study with. I would also like to thank our contact people at Debricked for answering our questions and providing quick and excellent support in using Debricked.

# Contents

# Chapter 1
# Introduction

In this chapter, we first give some background of the research area. Then, we describe the case company followed by definitions and terminology that is used in the thesis. Thereafter, the problem statement and its limitations are defined. Subsequently the contributions are described, and lastly the outline of this thesis report is described.

## 1.1 Background

Artificial intelligence (AI) could affect the software engineering (SE) field but within what aspects and to what degree is uncertain. This is partially because of that the software development tools using artificial intelligence today are new or in the development phase. Examples of recent software development tools using artificial intelligence are Amazon CodeGuru Reviewer and Tabnine [2] [68]. During preparation of this thesis most software development tools based on AI today turn out to be based on machine learning (ML), which is what we will limit ourselves to in this thesis. We therefore limit the scope of *artificial intelligence* in this thesis to be only machine learning. This thesis studies how software development tools based on ML can assist software development within programming, code review and bug management. In other words the thesis will investigate how the software developer and their work are affected by these ML-based tools, and what challenges the introduction of these tools might incur, along with an overview of what such tools there currently are.

The motivation to use AI within software development at a company developing software could be to achieve higher quality code or increased efficiency. By using AI in development tools the aim is to facilitate for both the developer themselves and to simultaneously bring financial savings and increased competitiveness for the company developing software. Before writing this thesis we hypothesised that development tools based on AI perhaps could lead to faster or more secure development, e.g. in the shapes of automatic code generation, less (or less severe) bugs, or automatic code reviews. Even if the financial savings only would amount to a few percentages of the software development department of a company, it could mean huge financial savings or increased competitiveness for a big company.

## 1.2 Softhouse

Softhouse is an IT-consulting company with more than 200 employees, where their office (Softhouse Consulting Öresund AB) in Malmö employs around 75 people. Softhouse is known to have worked a lot with introducing the agile way of working, e.g. they have certified thousands of scrum masters in the region. Most of their IT development business is as consultants or teams. Softhouse has six offices in Sweden and one in Sarajevo, Bosnia Herzegovina. It was founded in 1996. We will further on refer to Softhouse as *the case company*.

## 1.3 Definitions and Terminology

The definitions of abbreviations used in this thesis are defined in Table 1.1. Next, we define our terminology and definitions.

**Artificial Intelligence**. The term *artificial intelligence* (AI) has changed with time, and will likely be different in the future compared to its meaning at the time of writing this thesis [37]. The Oxford Dictionary of Phrase and Fable (2 ed.) defines *AI* as "*The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages*" [45]. This thesis limits the scope of AI to machine learning (ML), i.e. analysis of large amounts of data to find patterns that can be reapplied to new, unseen data.

**AI-based tool**. We define an AI-based tool as a tool within software development that uses AI. Two examples of AI-based tools are:

- Code completion feature to editors that gives suggestions to word completions while typing. There exist code completion extensions for editors, built on ML as well as conventional techniques. Here, AI could be used to possibly make better suggestions or a better ranking of the suggestions.

- Static code analysis tools. Here, AI could be used to perform a more "*intelligent*" code analysis more alike a code reviewer, e.g. by finding more complex patterns or variants of bugs, vulnerabilities or design problems.

**Table 1.1:** Abbreviations and their definitions.

| Abbreviation | Definition |
|---|---|
| AI | Artificial intelligence |
| CD | Continuous deployment |
| CI | Continuous integration |
| DWE | Digital work environment |
| ML | Machine learning |
| SE | Software engineering |

## 1.4 Problem Statement

The primary problem statement is defined as **How can AI-based tools that use ML assist software development within programming, code review, and identification & fixing of bugs?**. For simplicity *identification & fixing of bugs* will be referred to as *bug management*. As per the problem statement we limit ourselves to development tools within programming, code review and bug management. For example, software testing tools using ML will not be investigated. We decided to split the problem statement into three research questions.

- **RQ1: What ML-based software development tools for programming, code review and bug management exist today?** We will include a map of the ML-based tools found.
- **RQ2: How is the engineer's work and efficiency affected by AI-based development tools?** E.g. regarding:
  - Code quality
  - Code design
  - Bug management
  - Proportion of work time a programmer can save
- **RQ3: What challenges may introducing AI-based development tools in professional software development incur?** E.g. regarding:
  - Education
  - Work environment
  - Acceptance
  - Integrity (of the developer)

## 1.5 Contributions

The information about what AI-based tools that exist within software engineering today is scattered and our thesis could contribute by summarising what such tools there are (RQ1). The overview map can be seen in Chapter 4 and could e.g. be used as basis for what ML-based tools existed as of 2021. The map could lead to (or at least facilitate) a greater spread of knowledge within the subject, both about what AI-based tools within software development that already exist or to give inspiration about what tools that are yet to be invented. Our findings about how the engineer's work and efficiency is affected by AI-based development tools (RQ2) and what challenges introducing such tools within professional software development would bring (RQ3) could contribute to better (or faster) developed AI-based tools. This could be done by developers that develop these AI-based tools being aware of what challenges they incur, to improve them accordingly. This could in turn contribute to a reduced amount of time for AI-based tools getting a foothold within the professional SE industry, eventual standardisation, or to make AI-based tools within the software development work to be the new normal. Further research could also be performed on how to overcome the identified challenges AI-based tools are bringing. In addition, the work environment factors we studied to be able to answer RQ2 could be used for further evaluation of digital tools.

The thesis is also suitable to build future research on and could be seen as a future reconnaissance of how AI-based tools could change the software development profession in the future. Examples of future research could be further investigation of ML-based development tools, where our findings from the map or the impact on the developer could be used as a starting point. Future research could aim to to evaluate or further investigate some ML-based tools identified in the map. The map could also be built upon to include newer tools going forward or to include other areas the map excluded, e.g. testing. The map could also be used as basis for comparing ML-based development tools to other development tools, e.g. conventional tools or tools outside of ML but within AI.

## 1.6 Outline of Thesis

First, in Chapter 2 we bring up related work where we go through both conventional development tools as well as how AI has been used in software development previously. We also discuss how the work environment of developers might be affected when using digital tools. Then, the method used is described in Chapter 3. An overview and a map of what ML-based development tools there are is described in Chapter 4. Our results are presented in Chapter 5, followed by discussion about our findings in Chapter 6. In the end conclusions and possible future work are discussed in Chapter 7. References are added, and then there are also Appendices attached at the end.

To give the reader an overview of otherwise unknown literature reviews that we reference in the next chapter, we describe our methodology shortly next. The research method applied for this thesis work consisted of two literature reviews and a case study. The first literature review was used to explore the area of how AI is and has been used in software development, including what AI-based tools there are. The second literature review was performed in order to identify relevant work environment factors affected by development tools. In a field study we then investigated (mainly through interview questions based on these factors) how these factors were impacted by the use of AI-based development tools.

# Chapter 2
# Related Work

We have considered three main areas of previous research in order to be able to explore the impact of AI-based tools on software engineers, namely conventional development tools in general, AI-based tools specifically, and research on ergonomics and digital work environments. Firstly, our literature review on related work provides an overview of what traditional tools there are and how they are used in software development. Next, it also covers existing work on how AI has been used in software development. Finally, a second literature review on ergonomics & digital work environment is about work-related factors affected by digital tools and environments. In our work, these factors are used to investigate how AI-based tools affect developers and their work.

## 2.1 Conventional Development Tools

In software development there is a wide range of tools that assists developers in their tasks. In order to thoroughly be able to comprehend how AI has been used in SE it is a must, to a certain degree, to know what conventional development tools there are. Therefore, this section aims to provide a sufficient overview of the area. Next, we go through common types of development tools, closely related to the developer tasks programming, code review and testing, along with a few popular examples in each category.

- **Source code editor**, a text editor adapted to programmers, e.g. Atom [30], Visual Studio Code [54] and Sublime Text [41].
- **Build automation tool**, automatic compilation of software, such as Gradle [33] or Maven [27].
- **Integrated Developer Environment (IDE)**, a set of development tools that work together. Consists normally of editor, debugger, build & testing tools. Some examples are IntelliJ [42], Eclipse [24] and Visual Studio [53].

- **Version control system (VCS)**, management and versioning of changes to files. Some tools are Git [14], CVS [25], SVN [23] and Mercurial [49].
- **Source code repository**, web VCS service with additional features, such as facilitated collaboration, continuous integration, deployment pipelines or code review tools. Some examples are GitHub [31], BitBucket [5], GitLab [32], Azure DevOps [50] and AWS CodeCommit [62].
- **Static code analysis tool**, analyses code to find quality metrics, for instance code coverage (finding parts of a program not covered by test cases), or to find improvements like potential bugs or security vulnerabilities. A few examples of such tools are ESLint [26], Coverity [67], FindBugs [58] and Infer [21].
- **Testing tool**: Facilitates testing, and is usually a complete testing framework. Usually dependent on programming language. A few examples are Selenium [15], JUnit [72] and Spring [75].

In our work, we focus on programming tools and code review tools, since we are interested in how AI-based tools can assist *programming*, *code review* and *bug management*. Bugs can usually be identified through code review and corrected through programming. That, along with how *programming* and *code review* can be assisted, limits us to focus on *programming* and *code review* tools, each in its own subsection below.

We define programming tools as tools that helps the developers in their programming, like an editor, IDE or as a plugin to any of them. We define code review tools as tools that are integrated into a source code repository and normally includes bug management (including bug detection), with the purpose of assisting code reviews.

### 2.1.1 Code Review

For most software development teams and their projects, code review is an essential part of their work. Code reviews are performed on pull requests, which we will explain next. When a developer, whom we will call a contributor, has changed a piece of code (e.g. completed a feature) in his or her local branch (a branch is like a duplicate of some version of the source code project) and wants to integrate it with the rest of the code at the base branch of the online source code repository project, they can create what is called a *pull request*. This *pull request* can have none or several requirements that have to be met before the code changes can be merged in to the base branch. In professional software development there is often a requirement that one or more developers need to review and approve the code changes a contributor has made before it can be merged into the base branch. This requirement helps ensure that the code maintains good quality, and with such requirement a *pull request* can be interpreted as a *review request*, where the contributor asks the reviewers to perform a code review on the code changes. Thus, a pull request could be a way for developers to notify their team that they have completed a feature which is ready for review. Usually in professional software development a pull request needs approvals from all assigned reviewers, whom can either approve, discuss, or comment on any problem and ask for updates by the contributor to follow up with. The code changes are merged into the base branch if all reviewers approve the pull request. Although it is less rare to do so, reviewers can also themselves follow up with adding additional code changes to the pull-request.

The details of how thorough the reviewer reviews the code can vary. There can be multiple reviewers assigned to a pull request. Normally there are guidelines that tell the reviewers what to check for during a review. Guidelines could e.g. prompt a reviewer to check that certain coding conventions are being followed. The code review guidelines can range from being surface-level (or even non-existent) to in-depth, e.g. as checklists or documentation.

A large part of code reviews are performed solely through the web interface of the source code repository, which in a user friendly way provides the code change that, if approved, would be applied to the base branch of the online source code repository project. If there is a CI pipeline for continuous integration, usually the reviewer is informed if the project was successfully built and if tests passed. Normally a code review is performed without additional support from code analysis tools that ideally could inform the reviewer of potential bugs, improvements, or code and vulnerability issues. Some examples of code analysis tools are SonarCloud [65] and LGTM [61].

#### Social Impacts of Manual Code Review

Code reviews not only affects code quality but also the developers themselves. To be able to evaluate AI-based code review tools we need to take into account the benefits of the traditional code review, as performing code reviews may not necessarily be the same in the future.

Bosu et al. (2017) investigated the benefits of code reviews by surveying open source developers and developers at Microsoft who had taken part in at least 30 code reviews [13]. Their analysis showed that developers spend around 10 to 15% of their time in code reviews and the three most commonly reported reasons for performing code reviews they identified as maintainability issues, knowledge sharing and functional defects. Remarkably, the code author's reputation and the reviewer's relationship to the author were the two most dominant factors in deciding whether to approve a pull-request. The code quality reportedly affected the reviewer's impression of the code author in regards to skill level and personal characteristics. According to more than 75% of the respondents, poorly written code negatively affects their perceptions of the code author, possibly affecting future code reviews.

Impressions formed during code reviews could affect future collaborations or relationships. Bosu et al. mean that code reviews are critical not only to ensure code quality of pull requests but also for shaping the social foundation of successful projects.

### 2.1.2 Programming

Programming is the main task in software development. Some tools that can help especially during the programming part are features often included in many editors, such as automatic generation of code snippets, code completion, refactoring assistance, and linting (linting is automated checking of source code for programming errors, for example stylistic errors). The generated code snippets could be boilerplate code, class or method templates, or very common methods like getters and setters in Java. These features are not new, have long been existing and can be huge time-savers for professional developers, compared to manual repetitive work.

## 2.2 AI-based Development Tools

Our literature review revealed that AI in general, and ML in particular, is used in several types of development tools, e.g. editors, IDEs, and code analysis tools. With a surface-level review of conventional development tools, we can thoroughly grasp how AI has been used in SE. We present this research in chronological order next.

In 2002, Zhang and Tsai investigated the subject of applying ML in SE [79]. They noted that already by the 2000 shift, ML had been used within estimation of software quality and prediction of defects. They found that most works and tools using ML within SE at hand at that time, focused on estimating different project properties given the software, like quality, costs for e.g. development, maintenance, amount of required resources, time and work hours. Worth noting is that almost half of the publications studied by Zhang and Tsai were about how to build ML models for the purpose of estimating software properties. Furthermore, three ML techniques, namely artificial neural networks, case-based reasoning (part of instance-based learning) and decision trees, made up 61% (28) of all ML applications in their investigation. Perhaps these were the techniques most comfortable to use at that time. However, Zhang and Tsai remark that the existing works demonstrated that the SE field at that time was fertile soil for applying ML. The conclusion then was that ML has the potential to complement existing SE tools. The "intelligent" code review or code analysis performed using ML at that time was more about predicting if a file includes a bug or if it does not, i.e. binary and no actual *review* of the code, and instead about context, history and file properties. Zhang and Tsai did not mention any analysis of the source code itself or any variant of code completion using ML, adding the possibility that using ML in these parts was not actively discussed at that time.

In our literature review there is a lack of related work about AI-based development tools between 2002 and 2012. This could be due to Zhang and Tsai being early to investigate about applying ML in SE.

A paper by Ammar et al. (2012) investigated the application of AI to SE processes, which they remarked could reduce time to market and improve software quality [4]. The paper describes current state-of-the-art AI techniques and their applications in SE in 2012, along with current open research problems. As reported by Ammar et al. one of the hardest problems for AI within requirements engineering is to transform requirements into software. Perhaps currently a more feasible approach than solving this problem would perhaps be to use AI as an assistant rather than as a sole developer. Ammar et al. also mentioned that the automation in testing has also been done outside the AI area, using e.g. GA (genetic algorithms) to generate optimal test cases or by using CBT (constraint-based testing).

There is a lot of research on supporting fault detection in source code using AI. One example is a recent work (2018) by Ayesha and Yethiraj where they proposed a methodology for developing an adept code review system using ML [6]. They announced that as software was used in critical areas where faults may lead to disastrous consequences, numerous research had been investigated to detect faults in projects to improve software quality. The proposed methodology would feed file properties like author of file modification, the modification size, changed file metrics as well as source code metrics gathered from static code testing into a SVM classifier. However, a system implemented according to their methodology would only predict binary if a file is correct or needs rework, and not what the file specific issues would be. They state that fault prediction could be used for early detection of errors in order to e.g.

reduce costs or change testing procedures for error-prone modules.

A recent work (2018) by Nucci et al. investigated the applicability of using ML in detecting code smells - symptoms of poor implementation and design choices [19]. They invalidated the high performance results from a related study due to the datasets used which introduced great bias. Nucci et al. assert that the problem about detecting code smells is far from being solved, even if a lot of research has been put into code analysis using AI already, and that therefore there is a need of continued research.

The use of AI within software testing is another area that is under development and that is estimated to potentially be automated completely, according to a work (2019) by Hourani et al. which essentially performed a future reconnaissance about software testing using AI [40]. The authors are very optimistic in the paper and the authors assert that current accomplishments are very promising within most aspects of software testing, with a great future potential. Their optimism is well-founded on their literature review results, which they summarised with that AI had *"played a major role in software testing. Machine learning and NLP cover many testing areas..."*. In the paper, they describe many different types of testing and how AI could be used, which we refer the reader to for further details.

In 2020, Barenkamp et al. investigated how AI has been and potentially could be applied in SE [8]. They found that processing natural language into software code has been researched since the 1980s. Among the applications they identified, they found that defect analysis using neural networks outperformed classical testing routines. Fuzzing (testing invalid, unexpected or random data as inputs to a program) using AI also proved superior to traditional fuzzing techniques. Their analysis showed that future potential of AI are within, among others, the automation of prolonged regular tasks in software development and algorithmic testing, e.g. for debugging and documentation. Thus, AI helps speed up development processes and decrease development costs, according to Barenkamp et al. The major advantage that they see of automating SE practises, are that developers' creative potential increases substantially when AI tools are used in an efficient way. Current research, they point out, shows that neural networks have proven creative in practise tests, but the combination of reason, creativity, modalities and abstraction levels make developers far better than any machine today. Agreeing to that, Barenkamp et al. identified one survey which 328 experts took part in, where around 35% believed that human programmers will never be completely replaced in the testing phase by machines. Barenkamp et al. are very optimistic about AI within SE, meaning that it is a great disadvantage for software development companies to not use AI in their software development processes. They state that AI based SE might require software developers to be smarter or require additional competence in the future. They also remark that even if AI automation theories become debunked in the future, it still has the potential to speed up and assist software development processes.

Abubakar et al. investigated in 2020 the emerging aspects of the interplay between ML and SE for quality estimations, meaning how ML is and could be used in SE and vice versa, and possible applications [1]. Reportedly, estimating different project measurements and software properties is today extremely relevant. Their work covered many aspects outside or our scope, but related to our work is that they mention ML could be integrated in SE processes to enable automation among software codes and testing. They noted that agile development practises such as CI/CD cycles and agile iterations innovated conventional SE, while on the other hand code reviews by human reviewers in the CI/CD integrations are still necessary. According to Abubakar et al., ML recently was employed in SE to enable less coupling in legacy codes and between related modules, as well as automatic code versioning and refactoring. Different ML techniques are not always suitable, and have performed differently well within different applications of SE, Abubakar et al. added. They claim that generally ML has helped keep automation, adaptation, pace and scalability in pace to the increased development pace.

In 2020 Bilgin et al. investigated and performed vulnerability prediction from source code using ML [11]. Bilgin et al. comment that there now is an emerging interest in how to leverage ML in software development. In their study Bilgin et al. developed a source code representation method that enabled them to perform an "intelligent" analysis on the AST (abstract syntax tree) generated from the source code. ML was used to differentiate between which code fragments were vulnerable and which ones were not. The whole process they described in detail. Compared to an earlier work code2vec, Bilgin et al. achieved between 20 and 70% of accuracy, depending on the specific vulnerability evaluated.

For more recent progress with actual tools we refer to the map of AI-based tools in Chapter 4. A separate section for AI-based code review is omitted as the relevant works are provided as actual tools in the map.

Based on our literature review, AI-based tools within programming do generate some kind of code. It could be as simple as a code completion tool, which tries to suggest a word completion or the line that the user was typing at that moment. It could be as complex as trying to generate front-end code from an image representing the desired design. The result and categorisation of these tools is shown in the map in Chapter 4. One related study, already mentioned, is authored by Barenkamp et al. [8]. They performed a study with 5 participants, which worked with an AI-based code completion tool called Kite. One participant concluded that "*the tool reduced software development times and improved output quality*". Another participant mentioned that AI did speed up the debugging process.

Other than this study mentioned above, there was no other work found in our literature review where multiple participants used some AI-based development tools.

## 2.3 Ergonomics & Digital Work Environments

To be able to investigate how the work environment of developers is affected, a second literature review was performed (see *Planning* in Figure 3.1). In this section we start by summarising the resulting sources from most relevant to least relevant, about what seems to affect the work environment. We begin with a few major sources. These factors will further be used to conclude how the (digital) work environment is affected by AI-based development tools. We refer the reader to Table 2.1 for a summary of identified factors.

In one major source, [38], Helgesson et al. investigated cognitive load drivers in large scale software development. That is, factors that contribute to a cognitive or mental load on the developer. If these drivers induce a cognitive load on the user, Helgesson et al. state, these drivers are ideally kept at minimum, to give leave to practical work throughput or cognitive bandwidth for work. The two main clusters of load drivers concern the tool the *tool* itself, and the management, load, and flow of *information*. There is also a third smaller cluster that is about the *work process*. The *tool* cluster of load drivers in [38] contains the following drivers:

- **Intrinsic** which is about poor adaptability or suitability of a tool to work purpose, as well as lack of stability or reliability of a tool, which could prevent user trust in the

**Table 2.1:** Grouping of identified common (digital) work environment factors. The factors in bold is what we will refer to further on.

| Factor | References to topic |
|---|---|
| **Comprehension** | [35] [38] [77] |
| **Ease of learning**, learning curve | [35] [12] |
| **User satisfaction** | [35] [28] |
| **Training, support, documentation** | [35] [38] [12] [28] [16] [77] |
| **User control** | [16] |
| **System interplay**, integration between tools/systems, overlap | [35] [38] |
| **Work process interplay**, workflow, interaction with tool, ease of use | [35] [38] [12] [16] [20] |
| (Tool) **robustness, stability, reliability** | [38] [77] [16] |
| **Ease of remembering** | [35] |

tool if e.g. data is lost. *Intrinsic* also includes comprehension, i.e. understanding what is going on and if it is obvious how the tool should be used. *Intrinsic* also mentions overlap and lack of integration, which can prevent users from working efficiently, leading to redundant work e.g. having to copy information between systems.

- **Delay** as an absence of response in a tool can be viewed as forced mental concentration for the user when having to keep information in memory while waiting for the tool to respond.
- **Interaction issues** concern functionality that is missing or implemented in a way that makes the user have to spend unnecessary or redundant effort and time.

The second cluster of cognitive load drivers in [38] is about the load, flow and management of information, which is divided into *quality & integrity*, *organisation*, and *location* of information, which can be summarised as:

- Quality & integrity of information:
  - **Incompleteness**, causes effort asserting completeness of information.
  - **Lack of reliability**, causes effort asserting correctness and if it is up-to-date.
  - **Temporal traceability** over time (helps asses the current situation from what has previously been done, e.g. if a bug is reported or already fixed.
- Organisation (location) of information:
  - **Retrieval**, difficulties finding information costs user time and effort.
  - **Distribution**, where to put information or to whom to give it to
  - **Overview**, lack of overview or zoom views when browsing information

  - **Structure**, that is, if the information is organised in a clumsy or bulky way

The last cluster of cognitive load drivers in in [38] is about work process and can be summed up as:

- **Lack of support** which can result in wasted effort when spending time finding out how to perform something
- **Lack of automation** which can result in wasted effort.
- **Lack of understanding** of a project or of the fundamentals of an organisation

Another major source on work environment factors is a work done by Håkansson and Bjarnason, where they investigated what factors that are relevant to consider within requirements engineering that contribute to a healthier digital work environment [35]. Their focus was on how the user's interaction with an IT system affects work performance and psychosocial health. According to Håkansson and Bjarnason, job design, work load, mental load, and stress are all affected by the design and characteristics of an IT system, and the resulting factors they arrived at were categorised as *user support* and *system characteristics*. Under user support there are *documentation* and *training*. We will summarise the factors below. For further details we refer to their work, [35].

- User support
  - **Documentation** is an important factor in supporting users to use an IT system effectively. If it is missing or not updated, users do not know what information is correct and up-to-date. This could lead to frustration and work hours being spent by IT support answering the same questions repeatedly in how to perform certain tasks.
  - **Training** in how to use a new IT system is important for achieving or facilitating user acceptance of using it. Lack of training is one of the barriers for successfully deploying an IT system.
- System characteristics
  - **Understandability** refers to the degree a user understands an IT system's behaviour, how to use it, and how it works. Weak understandability may interrupt the work flow (when the user spends time and effort trying to understand how to use it) which induces stress and cognitive load. E.g. hard-to-find features or unclear error messages. Weak understandability can cause users to avoid or resist the new IT system.
  - **Ease of learning** in how to use an IT system for performing work affects time and effort the user needs to put in for work, and is related to *understandability*. Systems that are easy to learn and understood are preferable, as complex systems can be overwhelming for users and require months to learn. The latter can contribute to stress and decrease productivity.
  - **Ease of remembering** of how to perform a task is crucial when a task is performed with long intervals in between, and is related to *user satisfaction*.

– **User satisfaction** can imply that an IT system's users show lower levels of stress, compared to dissatisfied users. It has an impact on productivity as well as user acceptance of changes (a dissatisfied user may oppose changes). User satisfaction can be increased by providing *user support*.

– **System interplay** correlates with how well a user can switch between IT systems in their tasks, and the interaction between them, including sharing data. Poor integration between systems contributes to a cognitive load for the user, e.g. requiring a user to memorise or copy-and-paste data. This can affect user acceptance of an IT system.

– **Work process interplay** is a critical factor which is about how an IT system supports its users in their work. Rather than enforcing a certain way of performing work, good process interplay lets the user perform tasks in the order they deem best suited to the task at hand.

In a paper by França et al., they presented an investigation of what drivers software engineer's motivation and satisfaction at work [28]. As Hall et al. stated and França et al. quoted, the motivation of software engineers is "reported to have the single largest impact on productivity and software quality management, and continues to be undermined and problematic to manage" [36, p. 10:2] [28]. Among the results, França et al. present a new Theory of Motivation and Satisfaction of Software Engineers (TMS-SE), which converted and put well accepted theories together with new findings into the SE context [28]. TMS-SE can be seen in Figure 2.1.

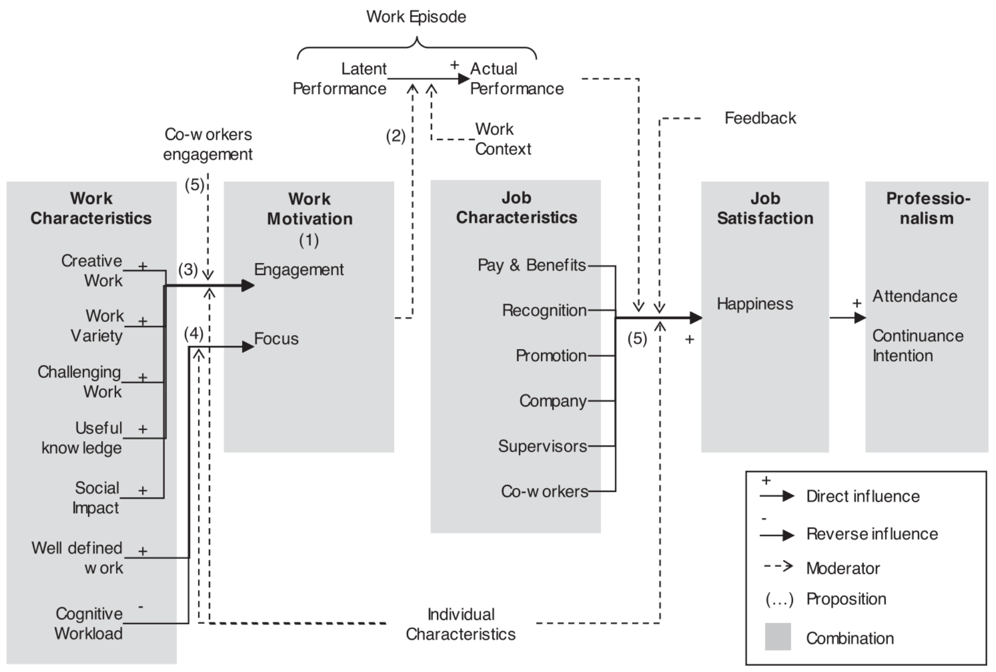


**Figure 2.1:** An overview of Theory of Motivation and Satisfaction of Software Engineers (TMS-SE) by Franca et al. [28].

The following cited works in this section are not major contributions but partly strengthens some of above sources. Wiggins et al. investigated how existing human factor related practises and models might support future human-centred technologies [77], such as development tools. They thought that "*The successful integration of new technologies in the workplace is likely to lead to a more productive and enjoyable interaction for employees and customers*". Three factors that Wiggins et al. identified several factors supporting the tool-user interaction (see Figure 2.2). Three factors are closely related to digital work environment, namely reliability (in technology), training (for users how to use new technology) and complexity (for comprehension, simplicity is preferred), which can be seen in Figure 2.2.

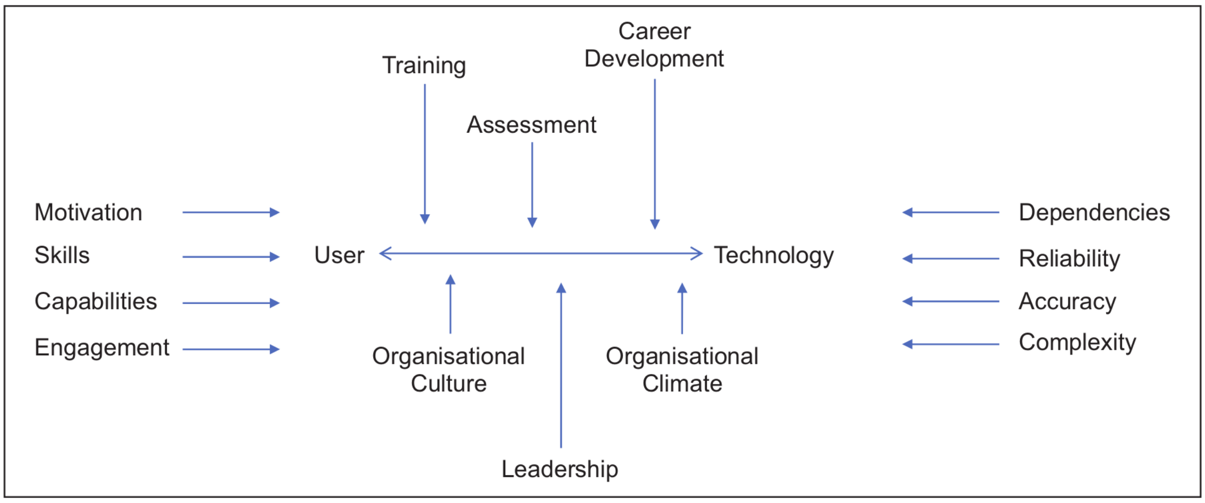
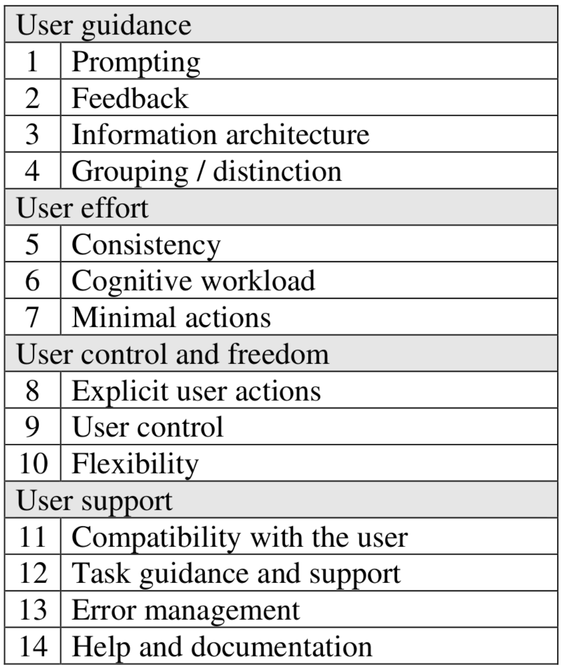


**Figure 2.2:** A picture by Wiggins et al. describing the integration between user contributions, technological capabilities and organisational influences necessary for optimal human–machine interaction [77].

Pribeanu analysed how to evaluate the usability of an interactive system and wrote a revised version of usability heuristics for interface design [16]. He builds his work on previous work targeting government websites. He attempted to improve earlier work, and stated that "*The underlying goal is to include the most important ergonomic criteria and usability heuristics into a clear hierarchical organization, which helps evaluators to better explain and developers to better understand the usability problems*". A usability problem was defined as "*any aspect of the user interface which might create difficulties for the user with respect to an important usability indicator*". A few examples that they provide are ease of understanding, learning how to operate, ease of use, time to complete a task, and subjective user satisfaction. The work goes into detail about each aspect but Figure 2.3 summarises the revised usability heuristics. We refer to [16] for detailed descriptions about each aspect.

Bordi et al. investigated the main digital communication factors affecting well-being at work [12]. The key relevant aspects they identified for our work are listed next. Firstly, large volumes of communication when using a tool is exhausting. Second, inadequate quality of messages had a negative effect in well-being as well as when one did not have allotted time for learning new communication tools. Finally, technical problems and negative user experiences e.g. lack of user-friendliness negatively affected workflow [12].

Dillon and Thompson proposed that tool usability may hinder efficient software development [20]. They also mention that for developer tools, it is the technology that should drive the requirements; development tools are designed from the bottom up, and it is not



| | |
|---|---|
| User guidance | |
| 1 | Prompting |
| 2 | Feedback |
| 3 | Information architecture |
| 4 | Grouping / distinction |
| User effort | |
| 5 | Consistency |
| 6 | Cognitive workload |
| 7 | Minimal actions |
| User control and freedom | |
| 8 | Explicit user actions |
| 9 | User control |
| 10 | Flexibility |
| User support | |
| 11 | Compatibility with the user |
| 12 | Task guidance and support |
| 13 | Error management |
| 14 | Help and documentation |

**Figure 2.3:** An picture by Pribeanu illustrating the revised usability heuristics [16].

uncommon for their tool usability to be very low. They also refer to work that suggests that source code editors often compete with the IDEs that were intended to replace them. They state that "*Consistent with the findings of various other researchers, we have shown that poor usability can force developers to use manual development techniques, avoid the use of tools and, in the case of novices, actually make code worse*" [20].

Lamb and Kwok performed a longitudinal investigation of how work environment stressors affect performance and wellbeing of office workers [46]. The stressors they brought up are about indoor environmental quality factors, such as office lightning, temperature, and noise. These stressors or factors are outside of our scope, but suggests there can be other external factors that could bias our results [46].

Lenberg et al. discussed human factors related challenges in SE through an industrial perspective [47]. Lenberg et al. pointed out that for software projects, customer relations is a factor crucial for product success, and that improving communication is considered an essential component. No other plausible work environment factors were found in the work authored by Lenberg et al. The paper did not identify factors but instead what themes of human factors that had to be considered within software development. The main themes are customer relations, organisational change, one-dimensional solutions, and communication. These themes concerned more about the organisation, its relationship to its customers and the relationship between internal employees [47].

# Chapter 3

# Method

The research method applied for this thesis work consisted of two literature reviews and a case study. The first literature review was used to explore the area of how AI is and has been used in software development, including what AI-based tools there are. The second literature review was performed in order to identify relevant work environment factors affected by development tools. In the field study we then investigated (mainly through interview questions based on these factors) how these factors were impacted by the use of AI-based development tools.

An overview of the methodology used can be seen in Figure 3.1. In order to find out how AI affects developers we had to make a map of what AI-based tools there are. This was needed because we did not find any related work that had mapped what AI-based tools there are in software development. As seen in Figure 3.1, the first part consists of a literature review of related work and AI-based tools. From the literature review we found candidates of development tools that might use AI. The tools that did not use ML were filtered out, and the remaining tools were further investigated to add them to our *Map of AI-based Tools* (as can be seen in Figure 3.1).

In the second part a case study was conducted in order to be able to answer our research questions. The planning phase of the case study consisted of a practical evaluation of which AI-based tools that were suitable to perform a case study with. In addition a project sampling had to be done in order to find available company projects and participants to perform the case study with. We also investigated different approaches as the available projects had varying amount of time set aside for this case study. Among the data collection preparations we identified certain survey and interview questions from company interests and a *literature review of ergonomics & digital work environment factors.*

Then, the data was collected through interviews, surveys and observations. This data was further analysed in order to get results. Each part in Figure 3.1 is covered by their own sections in this chapter.

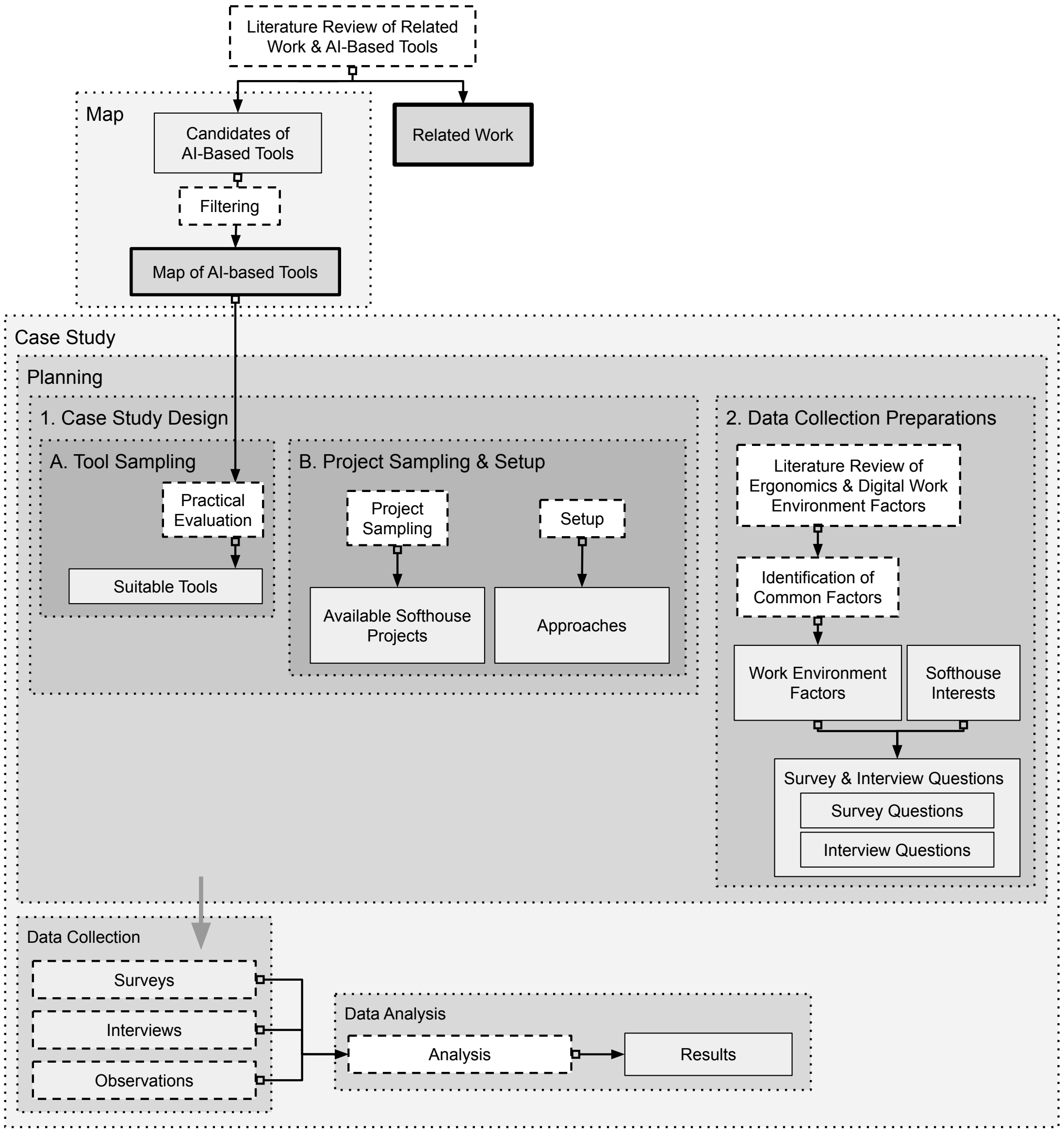


**Figure 3.1:** Method overview.

## 3.1 Literature Reviews

Two literature reviews were performed, one for mapping out the AI-based tools and one to identify related work regarding how digital tools affect the engineer's work. The literature reviews were performed with a book by Runeson et al. describing the master thesis process from beginning to end, used as a guide, and with the help of our supervisors [39]. The first approach we used was to use a combination of specific search terms and terminology.

In the first literature review we wanted to find all AI-based development tools as well as what previous work there was of AI for SE. The literature review engines used were LUB-SEARCH, Google and Google Scholar. For all of these, multiple search combinations were used. The search terms that follows were tried with and without prefix terms like **application of** or **utilisation of**, followed by terms such as **AI, artificial intelligence, ML, machine learn-**

**ing, NLP, natural language processing, AI-based tools**, with **and, in, for** following and ending with **Software engineering, SE, testing, unit testing, software testing, test case generation, programming, code analysis, code review, code completion, coding**. At this point in time we included words related to testing because we had not excluded testing yet, also we would otherwise risk missing out on tools for bug management. Other middle terms such as **built on** or **using** were also used, requiring some changes in the sentences. Some examples of complete search terms are: **AI in programming**, **Software engineering using machine learning** and **Application of software engineering using machine learning**. In order to not miss out on any AI-based tools the literature review was extended on Google with additional search terms such as: **Tools, AI-tools, ML-tools, assistant, assistance, automation, help, program, programs, AI-based**. When tools was discovered, their names also was included as search terms, e.g. **AI tools similar to ABC** if a tool named ABC was found.

For some of the relevant articles, the snowballing technique was applied for finding additional sources, i.e. forward citation tracking of (interesting, relevant) articles. Snowballing has proved to be an efficient method that could be used as part of literature reviews [59]. Found literature was continuously filtered away based on the title, then by abstract or by skimming through the text. Often literature that did not mention AI, ML or SE terminology could be discarded right away. The sources of found literature were selectively reviewed by their perceived relevance to AI for SE. Eventually multiple tools and sources had been encountered repeatedly, of which we were at a stage to begin creating the map of ML-based tools. Chapter 2 (Related Work) is based on this literature review and the map in Chapter 4 originates from the candidates found in this literature review.

For the second literature review, about ergonomics & digital work environments factors, LUBSEARCH was searched mainly, filtered on peer-reviewed sources. A lot of search terms included: **factors, human factors, human factors ergonomics, cognitive factors, psychosocial factors, happiness, wellbeing**. This, as complete searches or in combination with **remote work, office, office work, work, digital work, in the workplace, work environment, digital work environment, digital tool, IT system, code review, software development** (the latter of which was done as complete searches, too). The rest of the process of the second literature review were similar to the first literature review, it was mainly the search terms that differed.

The number of sources found and filtered away at the different steps for the first literature review (AI for SE including found AI-based tools) are estimated in Table 3.1. Out of the 95 relevant sources, 30 were web articles summarising a few tools like with a title similar to "*5 AI tools to use in programming!*". 40 articles were research papers related to either digital work environment factors or papers/articles about AI in SE or certain AI-based tools. 25 sources were straight web links to the homepage of AI-based tools. The estimations for the second literature review about digital work environment & ergonomics can be seen in Figure 3.2.

## 3.2 Making Map of ML-based Tools

The map of AI-based tools was designed based upon the outcome of the first literature review about AI for SE. This outcome had resulted in multiple candidates of development tools that potentially used ML. Firstly the candidate tools were filtered out by if they actually were dedicated to software engineering. More specifically a tool would have to be included within programming, code review or bug management. However, due to time constraints we were

**Table 3.1:** Estimated number of literature sources reviewed in the first literature review about how AI is and has been used in SE, including what AI-based tools there are.

| **Source filtering description** | **Number of sources (estimate)** |
|---|---|
| Literature filtered away by title | Not counted |
| Sources with relevant title | 145 |
| Sources still relevant after reading abstract and/or skimming through text | 95 |

**Table 3.2:** Estimated number of literature sources reviewed in the second literature review about digital work environment & ergonomics.

| **Source filtering description** | **Number of sources (estimate)** |
|---|---|
| Literature filtered away by title | Not counted |
| Sources with relevant title | 20 |
| Sources still relevant after reading abstract and/or skimming through text | 10 |

not able to investigate all candidates. The gathered candidate tools were then investigated further if and in that case also how they used ML and what it was trained on. If a candidate for being an AI-based tool did not describe any usage of ML it was filtered out.

Part ways through the investigation we realised that there was a too large number or AI-based tools to handle. The discovered field were unexpectedly large. A large part of the AI-based candidates regarded front-end web/app testing using AI to detect and test different wep/app elements (buttons, forms, etc.) using e.g. image and text recognition. Due to time constraints we had to decide whether we wanted to focus on these testing tools (within bug management) or the rest. We would still be able to cover most of RQ1 from the the problem statement if we constrained ourselves to not investigate the testing tool candidates further. Due to the previously mentioned time constraints (as we wanted to perform a case study after creation of the map) leading to us having to exclude some candidates, the map is not entirely complete We did however attach a list of unexplored products or websites that potentially may contain AI-based tools.

## 3.3 Case Study

We performed a case study according to the methodology by Runeson et al. where we wanted to investigate how the AI part in these AI-based development tools affect the developers [39]. At the company this thesis was written at there was several projects at our disposal with varying amount of work time set aside for this case study. The map of AI-based tools was not completed when we began planning the case study but at this point in time it was in an mature enough state to plan and perform a case study with a few of the identified AI-based tools. When there was available time, e.g. during data collection, we kept working on completing the map of AI-based tools.

## 3.3.1 Planning

As part of the planning stage, we designed the case study and performed some data collection preparations. Both of these are described in separate subsections next.

### Case Study Design

**Tool Sampling (A)**. The tool sampling consisted of a practical evaluation of promising tools, to deem which were suitable for use in the case study. We wanted to perform the case study with tools that ideally could be integrated into the normal day-to-day work flow of software engineers i.e. in this case during either programming or code review. We restricted ourselves in this case to focus on AI-based tools that might help during code review. We also wanted a tool that was maintained and up-to-date technology-wise. Due to time constraints we wanted a tool that was deemed not too time-consuming to set up for several case study participants. The criteria we used during tool sampling are that the AI-based tool should be:

- Meant for helping developers during code reviews
- Recently released, well maintained or actively developed, i.e. up-to-date technology-wise
- Available for use
- Quick to set up
- At least adequate documentation about setup and usage of the tool and its terminology explained
- Supports integrations with most popular online source code repositories (GitHub, BitBucket, GitLab, Azure DevOps, etc.)

We used these criteria for tool sampling of what code review tools we can practically evaluate in the case study. In order to find out what tools that likely would be quick to set up (unless it could be estimated) for case study participants, we tried setting up tools ourselves and used our own experience as that fulfilled all of the other criteria. For example if a tool only had a GitHub page with unclear setup and troubleshooting help, then the tool setup was not evaluated.

The sampled tools from the map of AI-based tools that fulfilled the criteria are:

- Amazon CodeGuru Reviewer
- Debricked
- DeepCode

These tools are further described in Chapter 4, Map of AI-Based Tools.

**Project Sampling & Setup (B)**. During project sampling our supervisor at the company talked to other project managers at the company who were interested or had available time in the project to participate in a case study. In addition, an e-mail was sent-out company-wide for seeking participants to the case study. For each project sampled we needed to know what programming language and online source code repository they used in order to know which of the sampled tools were feasible for each project. For example, Amazon CodeGuru Reviewer only supports Java and Python, it is therefore not feasible to use in a project which contains neither Python nor Java code.

Ideally in all of the projects we wanted to observe and get responses from people who used the tools in their workflow for a period of time. We found two suitable projects for this approach that we call the *code review* approach: project *GA* and project *CB*. In the *code review* approach the participants were supposed to use the tools assigned to them in their work flow for a period of time (e.g. two weeks). We also found additional projects that unfortunately did not have as much time that could be reserved for the case study. For these projects only a one-time usage seemed feasible, what we called the *code analysis* approach. The *code analysis* approach were deemed suitable for four different projects: projects *M*, *CF*, *H*, and *T*. Project *M* had more time than the other projects for the same approach but not as much to move it to *code review*, so we added a new approach that was something in between: the *repeated code analysis* approach, or in other words a shorter version of a code review approach. In the *code analysis* approach the tools were used and evaluated once by the participants. For the *repeated code analysis* approach the participants used the tool for a short period of a few days.

A summary of the sampled projects and the decided work task for the case study participants in each project can be seen in Table 3.3. In these projects we want to evaluate the sampled tools, which were Amazon CodeGuru Reviewer, Debricked, and DeepCode.

**Table 3.3:** Sampled projects and each of their case study participants' work task.

| Project | Work task |
|---|---|
| **GA** | Code Review |
| **CB** | Code Review |
| **M** | Repeated Code Analysis |
| **CF** | Code Analysis |
| **H** | Code Analysis |
| **T** | Code Analysis |

Unfortunately all of the sampled tools could not be used in all sampled projects. To be able to see if there are any projects at the case company were one or more code review tools could be included, additional criteria had to be set up. For each project, we also had to take into account that the tool could integrate with their:

- Programming language
- Online source code repository
- Project requirements

Another criteria that appeared for project GA and project CB was that the source code could not be accessed or transferred to any third-party. We noted that a challenge to get permission to perform this case study was that the people responsible for each project had to give permission of using these tools. A major point for approval in project GA and project CB were knowing how the project source code was going to be used or accessed by the tools.

These projects deemed it extra important to not risk give their source code away. Due to this, DeepCode could not be used for project GA and CB. The other projects approved the use of DeepCode. This limited us to solely use Debricked in project CB.

### Data Collection Preparations

We wanted to get answers about how the case study participants and their work were affected by AI-based tools (RQ2), which was the reason for the second literature review about digital work environment & ergonomics. From this literature review we identified common work environment factors between the different works, which were previously summarised in Table 2.1.

We prepared both interview and survey questions. The purpose of the survey was to get answers regarding the expectations of the case study participants of how they think AI ideally will affect their work. The purpose of the interview was to find out how the case study participants and their work were affected by AI-based tools. This would also allow us to later compare their statements during the interview with the original expectations of the participants. We used the *work environment factors* in order to be able to create good and thorough survey and interview questions for our case study. The survey and interview questions were created with the help of our supervisors. They were then improved over several iterations, again with our supervisors. After several iterations the quality of the questions was deemed good enough for our case study. We tried to keep the questions open to avoid bias and to be able to extract richer results from the interviews. All supervisors were discussed with in order to avoid bias.

We planned for an introduction and end survey as well as an interview in the middle and end of the case study for each project. Due to the short time the case study lasted with each project due to time constraints we had to change our planning to only use an introduction survey and an end interview for each project. The interview and survey questions can be seen in Appendix A and Appendix B, respectively.

## 3.3.2 Case Description

The project characteristics & setting are provided for the six projects below. A quick overview of the project characteristics can be seen in Table 3.4. The project setting and tools used can be seen in Table 3.5.

**Table 3.4:** Project characteristics overview.

| Project | Programming language | Source code repository | LOC | Project phase |
|---|---|---|---|---|
| **GA** | TypeScript | Azure DevOps | 58900 | Development |
| **CB** | Kotlin | BitBucket | 19900 | Scale-down |
| **M** | JavaScript | BitBucket | 38300 | End phase |
| **CF** | JavaScript | Bitbucket | 39100 | Development |
| **H** | JavaScript | BitBucket | 63300 | End phase |
| **T** | Java | GitLab | 63200 | End phase |

**Table 3.5:** Project setting and which tools are used. A ✓ marks yes, and a ✕ marks no.

| Project | Work task | Debricked | DeepCode | Amazon CodeGuru |
|---|---|---|---|---|
| **GA** | Code Review | ✓ | ✕ | ✕ |
| **CB** | Code Review | ✓ | ✕ | ✕ |
| **M** | Repeated Code Analysis | ✓ | ✓ | ✕ |
| **CF** | Code Analysis | ✓ | (✕) | ✕ |
| **H** | Code Analysis | ✓ | ✓ | ✕ |
| **T** | Code Analysis | ✓ | ✓ | ✓ |

## Case Company Interests

The core interests the case company had for this case study was to investigate whether AI-based tools were useful or valuable to them or the professional software development field. They wanted to get an insight if any specific tool would be interesting for them to use professionally, too. The case company aimed to gain an overview and understanding of what path the SE field will take, both present and in the future, in regards to AI-based software development tools. Focus was put on the psycho-social aspects of using AI-based development tools, e.g. if there are any cognitive advantages or disadvantages by using AI-based tools. A few specific questions the case company wanted answered, are:

- Are AI-based tools interesting for us to use?
- For a tool *X*, can *X* help us as programmers?
- How does AI-based tools affect the psycho-social aspects on its users?
- Can AI-based tools increase software development efficiency without any disadvantages?

## Code Review

When evaluating AI-based tools we wanted to investigate how they can assist code reviews. This code review approach was the ideal case, when both time and resources were available. We have two projects with this work task, namely project GA and project CB. The characteristics of these projects are provided in table 3.4, below are the background of the case study participants working in these projects. Each case study participant has an alias being the project name as a prefix, followed by a number (starting from 1). For example, if a project *X* would have two case study participants, these participants would be named X1 and X2.

**Project GA** consists of two developers, GA1 and GA2. GA1 has a software engineering education and has now worked at the case company for around a year, as a frontend developer. GA2 has a web developer education and has 4 years work experience at the case company. She started as a frontend developer but has now a frontend lead role.

**Project CB** is a project that has scaled down the number of active developers and consists now of one sole developer, CB1. CB1 studied a software engineering education, followed by working at the case company since 2017. CB1 has had different roles ranging from backend to full stack, in both big and small projects. The last year he has been working at project CB as backend developer, however.

### Repeated Code Analysis

**Project M** consists of one case study participant, M1. M1 has been working at the case company as a frontend developer for 3.5 years, in different types of assignments, ranging from 2.5 months to 18 months. He mainly works as a frontend developer but sometimes it's a mix of both frontend and backend (mainly react and node.js development).

### Code Analysis

**Project CF** has one developer, CF1. CF1 studied to Java developer and has worked on the case company for two years. However, his main programming language is JavaScript. and he is a full stack developer. He has worked a lot of frontend recently, so there is where the experience lies, he stated.

**Project H** has one case study participant, whom we call H1. H1 stated that he is not the most experienced, as he has only been working for two months at the case company after recently graduating, and worked in project H for about one month.

**Project T** consists of one case study participant, whom we will call T1. T1's current role is systems architect, but he is also a full stack developer, as he is developing quite a lot, so he does more than just designing systems, he stated. He has been a systems architect for 9 years, and has been a developer (including role as systems architect) for around 15 to 20 years.

## 3.3.3 Data Collection

Before the tools were introduced to the participants of each project, the introduction survey (shown in Appendix B) was sent out as a Google Form. We wanted the case study participants to be open and honest with their answers during the case study. To minimise the risk of them being dishonest, we therefore decided to have their identities be kept confidential and anonymous, and informed them of their anonymity.

While the case study was on-going we made a few noticeable observations, that we took into account during the data collection. An example would be if a case study participant needed extra help with something particular or there was troubles using any tool. These observations are also part of our results. The case study lasted approximately 2-3 weeks for each project between introduction and end interview. The interview questions in Appendix A were used as a template during these interviews. In our first interview, we discussed if the interviewees had learned or heard about the tool before they started using it. This was such a good question we had completely missed and not thought of previously, and we added it to our interview questions template for all the other interviews.

The interviews were recorded and later transcribed. The interviews were performed with one or two participants from each project. The interviews were generally performed with the participants that seemed to have used the tool(s) the most (and had available time to perform an interview, which luckily was the case).

### 3.3.4 Data Analysis

The interview transcriptions were coded in Microsoft Excel using thematic coding into the different work environment factors and then analysed. The results from this analysis, the survey responses and the observations can be seen in Chapter 5. The finding from these results were further discussed and connected to our research questions in Chapter 6 and concluded in Chapter 7.

# Chapter 4
# Map of ML-based Tools

In this chapter we present the map of AI-based tools. A lot of the AI-based tools found have similar characteristics. A few common denominators were found and we decided to split the tools into the categories *programming*, *code review* and *code generation*. In the *programming* category there are AI-based tools that can assist with code completion and in solving compiler errors. In the *code review* category there are tools that may help developers performing code reviews, either on the source code as a whole, per pull-request or branch basis, or per git commit. Often instant feedback is provided to the developer which could fix early detected problems before an actual reviewer reviews the code changed. Lastly, the *code generation* category contains tools that from a certain input, like images or text, generates (ideally) desired source code. An overview map of the ML-based tools found can be seen in Table 4.1. Note that Table 4.1 is meant to give an overview to the reader and not any details about the tools outside of their niche area. Next, we will go through the tools in detail, split into the subsections *Programming Tools* (Section 4.1), *Code Review Tools* (Section 4.2), and *Code Generation Tools* (Section 4.3).

## 4.1 Programming Tools

Programming tools are tools that in some way can help the developer to perform programming, in large part by generating or predicting small parts of code during programming. In the *programming* category we have five code completion tools and one debugging tool, these will be explained next, in their own subsections.

### 4.1.1 Code Completion Tools

Code completion tools are tools which are meant to complete what the user currently is typing, ideally more than one word at a time, sometimes whole lines or even small code snippets in some rare cases. A java example would be (containing a Pair class) completing

**Table 4.1:** Overview map of the ML-based tools. *Is further built (at least partly) from the tool on the row above it

| Category | Tool | Short Note/Description |
|---|---|---|
| Programming | Kite | Code completion |
| | Tabnine | |
| | IntelliCode | |
| | Stack Overflow AutoComplete | |
| | Aroma | |
| | DeepDelta | Debugs and solves compiler errors |
| | Graph2Diff* | |
| Code Review | CodeGuru Reviewer | Code analysis, full/per pull-request |
| | DeepCode | |
| | Embold | |
| | Infer | |
| | CLEVER | Code analysis, per git commit |
| | Debricked | Dependency vulnerability scanner |
| Code Generation | pix2code | Generates source code for web apps from images |
| | Uizard* | |
| | Sketch2Code | |
| | Fronty | |
| | Debuild | Generates React code from From a short text description |
| | DeepCoder | Generates a representation of a function from input-output examples |
| | PCCoder* | |
| | PROSE | |

`Pai` on a new line into `Pair pair = new Pair()` ending the cursor in the parenthesis so that the user can write arguments (or it intelligently fills them in). If the user declared variables previously to the statement, it is possible the code completion tool suggests the whole line completion including the arguments.

Because of the code completion tools being very similar, we describe the common characteristics between the commercial code completion tools. The most popular, seamless to use and well maintained AI-based code completion tools are Kite, Tabnine and IntelliCode. These three tools are very similar and there are only slight differences between then. These three tools are also very similar to conventional code completion tools in that the user experience might be near identical between them and the conventional code completion tools. The noticeable difference between AI-based and conventional code completion tools ideally lies in the accuracy of the code completion suggestions, and perhaps the installation process.

## Kite

It is clear that Kite uses ML (more specifically *deep learning*) as they state such on their website [43], however it is not entirely clear exactly how, except in that their ML models have trained on over 25 million files. Any more detailed or trustful information was not found regarding how Kite uses ML. One less than reputable source however mentions that for Kite the model is trained on the abstract syntax trees derived from the source code (which we deem sounds reasonable to assume) [78]. In the FAQ of Kite they mention that Kite does not upload any (or any processed forms of) code when it is used (unless Kite is deployed in a server and trained on a company's codebase) [44]. Kite reports varying number of supported programming languages on their website ranging between "*11*" in their FAQ to "*over 16*" on their front page. Kite works best on Python, all other programming languages offer a similar experience however, according to their FAQ. 16 supported editors are supported according to their front page and there exists both a free & paid version for Kite. Available for download at `https://kite.com/` [43].

## Tabnine

Tabnine is another code completion tool [68], which previously got acquired by Codota in late 2019 [76]. Codota had their own code completion tool (named after the same company, *Codota*) previously and their merged product is now called Tabnine. Tabnine supports 21 editors/IDEs according to their FAQ, whereas 18 are supported officially and three are supported through community developed plugins. Tabnine does not require any configuration to work, it is simply to just install the extension to the user's IDE or editor. Tabnine is available for download at `https://www.tabnine.com/`.

Tabnine uses up to three different ML models that work in tandem. The first model is trained on over one billion LOC of open-source code and can be run either locally or on the cloud. By default this model is run locally. When using this model on the cloud however, only the local context from your file is uploaded to be able to calculate the code completion predictions. The second and third models are trained and run locally only and learns from the user's (and optionally their team's) preferences, code selections and ongoing interactions. Tabnine explicitly state that they do not share any of the users' code (except local context of the code when using the first model on the cloud), and the second and third ML model mentioned previously is only kept private for the user and their team. Tabnine is available in three versions, a free basic version followed by the non-free versions pro and business. The first and second ML model are improved for the non-free versions. They also include specialised language models (currently only for Python, Go and Java) in their paid versions.

For ML Tabnine uses GPT-2 (Generative Pre-trained Transformer 2) for its deep learning models [70] [63] [57]. GPT-2 (created by OpenAI) uses the Transformer network architecture to implement a deep neural network. Tabnine itself is language agnostic (it supports all programming languages) but for its semantic (using programming language specific information) code completions it relies on other software implementing the Language Server Protocol, according to their FAQ [69]. A default range of language servers (can be overridden by the user) are included for many common programming languages. They support 25 different programming languages according to their front page.

## IntelliCode

IntelliCode, by Microsoft, is completely free and is only available for Visual Studio and Visual Studio Code, which are also developed by Microsoft [55]. In many cases IntelliCode is installed and enabled by default, but to make a user could visit the *Tools* settings in Visual Studio, and extension page (`https://marketplace.visualstudio.com/items?`

`itemName=VisualStudioExptTeam.vscodeintellicode`) for Visual Studio Code, respectively. For Visual Studio there are six supported languages for context-aware code completions, which they call "*AI-assisted IntelliSense completions*" (C#, XAML, C++, JavaScript, TypeScript and Visual Basic) while Visual Studio Code support five languages (Java, JavaScript, TypeScript, Python and SQL).

IntelliCode automatically generates its recommendations merged from the models it has trained. The base model is always included for the supported programming language that the user currently is using. This model is trained on (the top) thousands of open-source GitHub repositories. The user and their team also have the possibility to train one or more team models on the private code. The team model can be trained specifically on one computer or associated with a Git repository. Mainly IntelliCode "*is built around the GPT-C – a multi-layer generative pretrained transformer model for code, which is a variant of the GPT-2 trained from scratch on source code data*" [66].

IntelliCode also provides the feature of "*AI-assisted Refactoring*" i.e. suggested refactoring based in repetitive code edits or changes that the developer make [64]. "*To do so, IntelliCode uses an AI technology called program synthesis, and more specifically, programming-by-examples (PBE). PBE technology has been developed at Microsoft by the PROSE team and has been applied to various products to enable users to streamline repetitive tasks after providing a few examples*" [64].

#### Stack Overflow AutoComplete

Stack Overflow AutoComplete (2016) was just a pet project by a developer, trained on accepted answers at `https://stackoverflow.com` tagged with "*javascript*" and that had more than 50 points [60]. To quote the author, he trained on the dataset by "*walking the ASTs of those snippets and creating a "completion" fragment for each node, indexed by a fingerprint of the left-hand context*". When predicting code completions the author states that "*it uses the same logic to generate a fingerprint for the context at the current cursor position, and fuzzy-matches that against the database. Matching completions are sorted by a proprietary blend of post score, left-hand context similarity, and nearby identifiers*". There is no more information about how the tool works provided by the author [60].

#### Aroma

Aroma (2019) is a code completion tool made by Facebook that instead of completing one or a few words or a full line like other code completion tools instead completes whole code snippets [48]. Aroma was a research project however and not a maintained product. For the curious one their source code is available on GitHub along with documentation about reproducing the research results [22]. Aroma was trained on thousands of open-source projects. We refer to the paper for more information about Aroma [48].

### 4.1.2 Compiler Error Solvers

#### Graph2Diff

Graph2Diff is a recently (2020) developed deep learning architecture, developed at Google, that automatically attempts to fix compile errors (also known as *build errors*) using ML [71].

Graph2Diff is able to do this from analysing the source code, configuration and compiler error of the program. It works by attempting to localise the issue and repairing the program by suggesting a *diff*, which they describe as *"a sequence of edit operations that can be applied to the broken AST to obtain the fixed AST"*. For a domain-specific language the Graph2Diff networks map a graph representation of the faulty source code to a diff that describes the repair. In the paper, the model was trained on a dataset with 500k fixes [71]. Graph2Diff showed to be 26% accurate on exact diff to solve a problem, which they compared to the previous most closely-related work *DeepDelta* (2019) they could find, which achieved 10% accuracy for predicting less precise diffs. Except for the paper and the preprint version, no other information or source code for reproducing the results were found, i.e. this tool is not officially available to the public.

#### DeepDelta

*DeepDelta* (2019), also developed at Google, attempted to partially solve compiler errors as well using ML. The paper about Graph2Diff also summarised DeepDelta well, *"DeepDelta system aimed to repair build errors by applying neural machine translation (NMT), translating the text of the diagnostic message to a description of the repair. Although this work is promising, the use of an off-the-shelf NMT system severely limits the types of build errors that it can handle."* [71]. It also stated that compared to Graph2Diff, however *"DeepDelta can only be evaluated on a less-stringent task than exact developer fix match (because it has no pointer mechanism to precisely specify locations)"* [71]. The accuracy compared to Graph2Diff is described in subsection 4.1.2. In *DeepDelta* just like for Graph2Diff the source code for reproducing the results is not accessible and it does not seem to be available to the public.

## 4.2 Code Review Tools

Code review tools have the characteristics of that they alert the user about probable code issues. They could detect things such as general code issues, security vulnerabilities, unnecessarily expensive lines of code executed, bad programming practises, and design or architecture related matters.

### 4.2.1 Amazon CodeGuru Reviewer

Amazon CodeGuru Reviewer is part of the tool Amazon CodeGuru (which also contains a runtime profiler outside of our scope) and is a static code analysis tool by Amazon that detects defects and deviations from best coding practises for Java and Python [2] [3]. It can perform either a full code analysis or analysis of a pull-request. The analysis result provides the developer with a list of *"actionable recommendations for improving code quality"* [3]. It support Java and Python stored in either GitHub, Bitbucket or AWS CodeCommit. According to their FAQ, *"Amazon CodeGuru Reviewer checks for concurrency issues, potential race conditions, un-sanitized or malicious inputs, inappropriate handling of sensitive data such as credentials, resource leaks, and also detects race conditions and deadlocks in concurrent code. It also suggests AWS, Java and Python best practices and detects cloned code that could be consolidated for better code maintainability"* [3]. Amazon CodeGuru Reviewer does not store any copy of the source code, but they do need to upload it temporarily in order to perform an analysis. Amazon CodeGuru Reviewer is a paid service but includes a free 90-day trial for analysis of up to 100k lines of code.

Amazon CodeGuru Reviewer does not solely depend on ML as they state that "*CodeGuru Reviewer uses machine learning and automated reasoning, AWS and security best practices, and hard-learned lessons across millions of code reviews on thousands of open-source and Amazon repositories to automate code reviews.*" [3]. In the FAQ of CodeGuru Reviewer it is explained that CodeGuru Reviewer is trained using supervised ML models that use a combination of logistic regression and neural networks. For resource and sensitive data leaks, it uses logistic regression models and convolutional neural networks. Feedback provided by a code reviewer (in the form of labels) to code recommendations by the tool is used to iteratively improve the product [3]. Amazon CodeGuru Reviewer (part of Amazon CodeGuru) is available at `https://aws.amazon.com/codeguru/`.

### 4.2.2 DeepCode

DeepCode is the predecessor of Snyk Code and is another static code analysis tool that alerts of critical bugs or security issues either by full analysis or automatically on every pull-request. In each issue detected, DeepCode shows a few diff examples of how other developers in open source projects have solved the same issue previously. DeepCode is trained using ML models on thousands of open source repositories. DeepCode was available at `https://www.deepcode.ai/` but was aqcuired by Snyk earlier this year. Now the corresponding tool is named Snyk Code.

### 4.2.3 Embold

Embold is a static code analysis tool for building high quality software. Gives detailed visualisations about the existing issues in a software project regarding best coding practices, design anti-patterns that makes the code unmaintainable. During the case study, Embold did not contain any ML

### 4.2.4 Infer

Infer is a static code analysis tool made by Facebook, only available through the command-line, which can detect potential bugs for six different programming languages [21]. For more information about Infer, we refer to [21].

### 4.2.5 CLEVER

CLEVER (2018) is a risky-commit detector made by Ubisoft and Mozilla [56]. CLEVER uses ML to discover files that contain one or more bugs with an average accuracy of 75%, and was used at a game development company called Ubisoft. This product was company specific and depended on detailed version management and tracking of bugs in earlier releases in the same project. CLEVER is not accessible to the public. For more information about CLEVER we refer to [56].

### 4.2.6 Debricked

Debricked is a security focused SaaS (software as a service) tool which identifies (package) dependency vulnerabilities from multiple sources such as NVD, GitHub issues, etc [17]. Debricked can be integrated through many different integrations, e.g. through online source code repository pipelines, through command-line or simply by logging on their website through e.g. GitHub or BitBucket. Gives a complete view of a software project's vulnerabilities and licenses [17].

Debricked uses some specific terminology in their product. It uses different kinds of scores for each vulnerability, CVSS2, CVSS3 and an AI-based vulnerability score that they call debAI [17]. The user can put limits for these scores for Debricked to warn, fail the pipeline or send an e-mail when vulnerabilities have been detected exceeding the current pipelines. Debricked do not use sensitive source code when performing a vulnerability scan, instead it solely transfers a converted version of the dependency package file (e.g. package file used by npm or yarn package manager) [17]. The ML it uses comes from collecting and interpreting huge amounts of data, e.g. GitHub issues, NVD, vulnerability databases, and more (E. Wåreus, personal communication, February 25, 2021).

## 4.3 Code Generation Tools

### 4.3.1 pix2code

pix2code is a recent research project (2017) that uses ML to convert images to source code [10]. "*pix2code*" is not misspelled, the authors of the research project named pix2code without a first capital letter. The motivation of the research paper was that implementing GUI code and interfaces are time consuming, the authors investigating if ML could help us. The source code of pix2code is freely available at `https://github.com/tonybeltramelli/pix2code` [9]. As stated on their GitHub page, pix2code is only a research project and will therefore not support any other target platforms than Linux.

The ML used in pix2code to generate code from a given image is based on deep learning, specifically it is based on convolutional and recurrent neural networks. They have no feature extraction pipeline, the model simply learns from the pixel values of the images alone, which in large is thanks to the convolutional neural network [10]. Different models were trained to be able to generate code for several target platforms being Android, iOS, and web (HTML/CSS). They were also trained on different datasets as a the datasets are not compatible with each other, e.g. iOS and Android. The dataset trained on is also available at their GitHub page (at `https://github.com/tonybeltramelli/pix2code/tree/master/datasets`) so that anyone can reproduce the results. On their GitHub page they state that the official research page of pix2code is `https://uizard.io/research#pix2code`. Notice that the domain is `https://uizard.com`, Uizard is a commercial tool we will talk about next.

### 4.3.2 Uizard

Uizard is a recent (2018) commercial in-web-browser design editor meant for creating design mockups in minutes [73]. One of the features of Uizard is to upload an image, e.g. a photo of a web interface sketch, to generate corresponding design in their interface designer. Uizard is available at `https://uizard.io/` and any design can be exported to source code for several frontend frameworks, e.g. React. The user is in complete control whether they want to use a design, fine-tune it in their online design editor or export it as-is. However, in order to export any design to source code however, a user needs to use the paid version of Uizard.

Uizard is based on *pix2code* among additional research and scientific contributions, i.e. at least the same ML technology that *pix2code* is built on. It is hard to say exactly what their export to source code feature is based on, however they describe the different technologies they overall at `https://uizard.io/research/` [74]. Most research papers they refer to are very recent (0-3 years old). The export function seems to be based on pix2code or an improved version of it. Potentially newer ML technologies will replace pix2code onwards, e.g. as they state by creating a GAN [34].

### 4.3.3 Sketch2Code

Sketch2Code is a tool owned by Microsoft and available at `https://sketch2code.azurewebsites.net/` [52]. From an image or a photo it tries to predict the HTML code that corresponds to the certain image. Sketch2Code is free to use with no registration required, anyone can use it directly on its website. For more information about Sketch2Code we refer to [52].

### 4.3.4 Fronty

Fronty is another image-to-HTML tool just like Sketch2Code, however its output uses HTML in combination with Bootstrap [29]. For more information about Fronty we refer to [29].

### 4.3.5 Debuild

Debuild is a tool that from a given input text can estimate frontend React code [18]. It is build using GPT-3. An example input text is *"A large text that says 'WELCOME' and a blue button that says 'Subscribe'"*. For more information about Debuild we refer to [18].

### 4.3.6 DeepCoder

DeepCoder is a research project by Microsoft that using ML from given input-output examples (think *test cases*) to predict what a function should do [7]. A free unofficial GitHub implementation is available at `https://github.com/dkamm/deepcoder`. For more details about DeepCoder we refer to [7]. An example the paper [7] provides however is where the input is two lists of integers [7 3 8 2 5], [2 8 9 1 3] and output being and integer 79, then the predicted program by DeepCoder is:

```
x <- [int]
y <- [int]
```

```
d <- sort x
d <- sort y
e <- reverse d
f <- zipWith (*) d e
g <- sum f
```

### 4.3.7 PCCoder

PCCoder is basically an optimised version of DeepCoder (and has its own research paper) with better implementation notes for reproducing the results [80]. The official implementation of PCCoder was built on the unofficial implementation of DeepCoder, and is available at `https://github.com/amitz25/PCCoder`. For more information about PCCoder we refer to [80].

### 4.3.8 PROSE

PROSE is a tool similar to DeepCoder and PCCoder in that it from input-output examples can estimate a function [51]. It is not open source but Microsoft has provided a few SDKs to be able to use it, and it is already included in some of their products like Microsoft Excel. For more information about PROSE we refer to [51].

## 4.4 Left Out Tools

Due to time constraints not all tools were able to be investigated. We have compiled a list of interesting tools, companies and websites that might have AI-based development tools, however we have not been able to investigate them further. Some tools were also discovered after the case study began, of which we did not have time to investigate them either. Below is a list of tools that might use AI.

- Sapienz
- Getafix
- Sapfix (Supposedly uses Sapienz and Getafix together)
- CraneAI (Project planning and testing)
- WhiteSource
- Tools made by MicroFocus (like Fortify Software Security Center & Fortify Static Code Analyzer (Code analysis, eliminates false positives using AI?))
- Tools made by Smartbear
- Prophet
- AIDA

- (Mendix) Assist (Code completion for configuring microflows)
- accessiBe (web compliancy)
- Commit Guru
- Some products by SeaLights (Bug management and test coverage)
- Tools found at `https://gpt3demo.com/`
- ReportPortal, `https://reportportal.io/`
- Test.ai, `https://www.test.ai/`
- Tricentis Tosca, `https://www.tricentis.com/products/automate-continuous-testing-tosca`
- GitHub Copilot
- ControlFlag by Intel Labs
- Cover by Diffblue

# Chapter 5
# Results

Our results are derived from the interviews, the observations, and the surveys (see Chapter 3 about method), and are described for each of the three investigated development tasks, i.e. code review, repeated code analysis, and full code analysis. An overview of the survey responses can be seen in Table 5.1, whereas we refer to Appendix B for the detailed survey questions and possible answers. We provide a part of user satisfaction below where each interviewee were asked to rate the tools they used from 0 to 100 overall. These user ratings can be seen in Table 5.2 for Debricked, and in Table 5.3 for DeepCode. For Amazon CodeGuru Reviewer there was only one user that could provide a user rating, see Table 5.4.

## 5.1 Code Review

The results from the case study are presented per project, and then per work environment factor. As there are multiple projects with different tools and work tasks that we analysed the work environment factors for, we had to take into account how to present our results in a good way. After discussion with our supervisors, we decided that presenting the results per project and then per factor would let us present the results in a more comprehensive way than other structures would.

Each participant in the case study is named after their project and a number as suffix (starting from 1). For example, the two participants in project GA are called interviewee *GA1* and interviewee *GA2*. Each participant keeps the same name consistently throughout the thesis.

### 5.1.1 Project GA

Project GA consisted of two interviewees GA1 and GA2. Their work task was *code review* (see Table 3.5 on page 30) and they evaluated Debricked. Both interviewees participated in the interview.

**Table 5.1:** Introduction survey results for the non-text survey answers.

| Project | Participant | How much time do you spend on one pull request in average (minutes)? | What would you rate the overall code quality of [Project]? | To what degree do you expect AI-based tools to affect your code review productivity? | To what degree do you expect AI-based tools to affect the quality of your review work? |
|---|---|---|---|---|---|
| **GA** | GA1 | 6 | Good | Simplify | Increase quality |
| | GA2 | 15 | Good | Simplify | Increase quality |
| **CB** | CB1 | 20 | Good | Simplify | Increase quality |
| **M** | M1 | 15 | Fair | Simplify | Increase quality |
| | M2 | - | Fair | Simplify | Increase quality |
| **CF** | CF1 | 10 | Fair | Simplify | Increase quality |
| | CF2 | 5 | Fair | Simplify | Increase quality |
| | CF3 | 5 | Fair | Complicate | Increase quality |
| **H** | H1 | 120 | Excellent | Simplify | Neutral |
| | H2 | 20 | Fair | Simplify | Neutral |
| | H3 | 10 | Good | Simplify | Neutral |
| **T** | T1 | 8 | Good | Simplify | Neutral |
| | T2 | 1 | Good | Neither | Increase quality |

**Table 5.2:** Debricked user rating.

| **Participant** | **Rating** (0-100) | **Why?** |
|---|---|---|
| **GA1** | 75-80 | It covers their needs, but minus for the user interface and understanding of how to intuitively use Debricked |
| **GA2** | 80 | For the same reasons as GA1 |
| **CB1** | 65 | The interviewee had not had time to use the tool. What drew it down was that Debricked could have been clearer. Did not feel he could rely on Debricked |
| **M1** | 65 | Debricked required planning to be helpful in the project. Project M was not well maintained or structured and Debricked was a bad fit there. Long delay encountered without feedback during pull request generation |
| **CF1** | 90 | Security wise 100, what drew it down to 90 was the user friendliness, specifically that there was no feature to see what parent dependency that uses a vulnerable dependency |
| **H1** | 90 | Well-made, comprehensive visuals, well-structured interface, smooth integration, and it does everything in the background by itself |
| **T1** | 75 | Easy to get going, smooth to use, simple, and has a clear purpose |

From the introduction survey, interviewees GA1 and GA2 were asked how they expected

**Table 5.3:** DeepCode user rating.
** Estimated rating.*

| **Participant** | **Rating** (0-100) | **Comment** |
|---|---|---|
| **M1** | 50* | Not-so-good suggestions, the issues and explanations detected by DeepCode were too general and not specific |
| **H1** | 60 | Well-made, comprehensive visuals, great potential but for this project DeepCode did not show any real strength which drew it down from 90 to 60 |
| **T1** | 80 | Exceeded his expecations, found his planted bugs, one drawback of needing to transfer source code to third-party |

**Table 5.4:** User rating for Amazon CodeGuru Reviewer.

| **Participant** | **Rating** (0-100) | **Comment** |
|---|---|---|
| **T1** | 80 | The exact same comment as T1 left for DeepCode in Table 5.3 |

an AI-based code review tool to affect their work, e.g. regarding quality and productivity. Interviewee GA1's response was "*Catch issues I miss and save time by pointing out issues before I need to go through the code*" while interviewee GA2's reply was "*Detect redundancies in logic*".

**Training, Support & Documentation**. Both interviewee GA1 and GA2 had previously heard about Debricked, but had otherwise no prior knowledge about it. They perceived our training (introduction) to have been good, and to have been to the best of our abilities at the time, possibly also because they knew that this was our first introduction and that we had not worked with Debricked ourselves. They remarked that our introduction was missing the vulnerability scores that Debricked used, how to set up the BitBucket pipeline, and what reasonable pipeline rules to use in Debricked were.

Regarding support, the pipeline was set up with interviewee GA2 and the help of us, and we had help of the documentation and previous setup experience. Interviewee GA2 also received support through the chat function of Debricked, when she had problems creating an account. It turned out that Debricked complained because an account with the company she was employed at already existed, according to interviewee GA2. She stated that they responded to her e-mail and resolved her issue within a timely manner. However without knowing the reasons for this obstacle, we deem Debricked to have been easier to get started with without such restriction. It could possibly be a barrier for new customers to Debricked if different people without each other's knowledge at the same company were to try out Debricked.

The interviewees used the documentation to read about scoring values and pipeline rules, and interviewee GA2 described it as "*probably pretty good*". They really appreciated that the documentation contained an example of default vulnerability scores to use as limits among the pipeline rules, which GA2 said were good for people, like them, whom are not security experts, to get started in using Debricked. However interviewee GA2 also stated that it would have been great if there was any documentation existing about solving dependencies to dependencies, of which they had the most problems with.

**Learning**. The interviewees mentioned that when learning to use Debricked, they cooperated and helped each other a lot with understanding and using it. Interviewee GA2 stated that she initially "*did not understand anything*" about Debricked, and further explained that it was new terminology and a completely new tool to set up. Interviewee GA2 told us that especially the pipeline rules were hard to learn, when and how to set a rule to trigger, and what vulnerability scores to use as limits. Interviewee GA1 had also troubles learning about pipeline rules, and explained further that even if it was quite obvious how Debricked was supposed to help them, it was still not comprehensible when looking at the interface for the first time, especially the overview which showed a list of all found vulnerabilities. Interviewee GA2 agreed. Interviewee GA1 further explained about Debricked that "*It had just gone and found these issues, then you just get this list, your packages, with some number on the side. It is not very understandable in the beginning*". However, interviewee GA1 asserted "*As soon as it starts to click in about what it is about these packages, with this and that issue, and what the next step is and so on, then it began to give me more meaning*". Interviewee GA2 added that the overview list was hard to grasp because most vulnerabilities had *not* to do with the direct project dependencies, but the sub dependencies to the project dependencies. She asserted that it would have been easier to know how to use Debricked if there was documentation to solve these sub dependencies, as neither interviewee GA1 nor interviewee GA2 said they understood how to fix vulnerabilities for such sub dependencies in the beginning. As soon as she understood what packages to update and how to update them in order to solve the vulnerabilities, using and learning Debricked was easy, interviewee GA2 affirmed, following up with "*because then it was to simply just update the packages and make a new pull request, and everything is solved*". Related to the unintuitive interface navigation interviewee GA1 mentioned, interviewee GA2 pointed out that in the beginning it was just unintuitive finding where in the interface to set up these pipelines rules, but after navigating Debricked for a while it became easy, she added. Another confusing thing GA1 said that he encountered when learning to use Debricked where when there were multiple vulnerabilities detected by Debricked for the same package. However, when solving one of the vulnerabilities often most or all of the other vulnerabilities for the same package was deemed as fixed too by Debricked, according to him. Lastly, interviewee GA2 not being able to create an account when an account for the same company already existed, as previously mentioned in *Training, Support & Documentation*, seemed to be an unnecessary obstacle when starting to use Debricked.

**Work Process Interplay**. Project GA integrated Debricked into their work flow. First, they solved all the vulnerabilities detected by the tool and then set up suitable default pipeline rules. Interviewee GA1 and GA2 described that when a new vulnerability appeared, the pipeline would fail and they would get an e-mail notification about it. Nowadays they only look if the pipeline complains and they get an email, both interviewees GA1 and GA2 clarified. In their normal day-to-day work, the interviewees mentioned that when they push changes to a branch they can see if it went through the pipeline or if anything went wrong. If the pipeline fails, they look into Debricked, see the vulnerable package, create a new branch, update it, and then merge the vulnerability fix changes in to the base branch, the interviewees commented. In the case study this had occurred between every few days and around once a week, they the interviewees added.

**System Interplay**. Project GA already had an existing pipeline which included at least static code analysis by SonarCloud and testing, according to GA2. Interviewee GA1 and GA2 also mentioned using linting tools locally during development. When interviewee GA2 integrated Debricked in project GA (with some help from us), we observed that she could not get it to work in the existing pipeline (in Azure DevOps), which had a different pipeline configuration than the default one Debricked provided in the documentation. Interviewee GA2 ended up creating a new separate pipeline with the default pipeline setup copied from the documentation of Debricked, which worked. She remarked that unlike attempting to integrate Debricked to their existing pipeline, creating a new pipeline for integrating Debricked was not hard at all. Then, they set up the pipeline behaviour from Debricked interface to fail the pipeline and to send an email alert to both interviewee GA1 and GA2 if any severe vulnerabilities were found. The interviewees pointed out that if there would not be any integration to their pipeline to alert them, they would *not* have periodically checked if any new vulnerabilities had appeared. The interviewees added that if they had time available for it ahead, they would look into integrating Debricked to their primary pipeline.

**User Control**. The interviewees mentioned that the existing settings and customisation of Debricked covered their needs. Interviewee GA2 stated that it was easy to use and integrate Debricked as preferred by its users since it support multiple means of integration, e.g. pipeline, command-line, manually through the interface, etc. GA2 stated that they were using the recommended or default rules from the documentation. The default rules were suitable for people without expert knowledge in security, according to the interviewees. It was fully up to the user what to do with the discovered vulnerabilities, too, usually from old packages, according to the interviewees.

**Comprehension**. After learning how to use Debricked and its terminology the interviewees thought that its detected vulnerabilities began to make sense. Interviewee GA1 pointed out that the overview which is a list of vulnerabilities was not obvious to understand, however he thought it was comprehensible when navigating to each vulnerability individually. In the overview interviewee GA1 mentioned that there were just a list of vulnerabilities with some packages and numbers, which he said at first could be very confusing. Interviewee GA2 agreed that the overview list was hard to understand.

According to interviewee GA2 they found it confusing for them when Debricked detected a vulnerability for a dependency that they did not have directly. The interviewees further explained that after investigating a vulnerability, the case was often that it affected a sub dependency to a direct dependency that they had. Debricked did however not give any indication of that it was a sub dependency and not a direct dependency, and their documentation did not give any hint on how to solve vulnerabilities affecting these sub dependencies, according to the interviewees. We think that for project *GA* the comprehension for Debricked would probably had been higher if there was a feature to see the what direct project dependency that a vulnerable sub dependency has as parent.

Debricked had a feature to mark a vulnerability as *unaffected*, meaning the vulnerability did not affect the project. Both interviewees GA1 and GA2 thought that it was unclear when to use it, and affirmed they had not used it at all. GA1 explained that if a vulnerability is detected then you just fix it. It seemed like all the dependencies that project *GA* had were maintained or well chosen for them to not having had encountered any discontinued packages without any vulnerability fix. GA2 commented however that they might have used the feature temporarily if a the vulnerable package did not have any update (that would fix the vulnerability) or there were no substitute package. For us, we deem it seemed like for interviewee GA1 and GA2 it was actually easier to fix a vulnerability than it would be to figure out if it actually did affect the project or not. Even if a developer would figure out that a vulnerability did not affect their project, we ask ourselves if the average developer would be completely sure about that. Even if he or she would be sure, perhaps it could affect the project in the future. These might be some reasons that interviewee GA1 and GA2 thought that if you were able to fix a vulnerability, then the best way to handle it would be to fix it instead of putting effort into finding whether it does affect the project or not. This might perhaps not be the case for other projects where each vulnerability fix would lead to an significant increase in work hours spent, e.g. on testing, where it could be of lower cost to examine whether a vulnerability actually affects the project.

Previously under the *Learning* factor, interviewee GA2 described that it was hard to find where to set up these pipeline rules. Related to comprehension this seemed to be something she just had to learn, because after learning Debricked she stated that it was easy to found. However it might have led to an increased mental load when learning to use Debricked, especially when learning to set up pipeline rules. We deem that in order for Debricked to improve their interface, one task would be to make this feature more easily found in the interface. Another thing that could have added to some mental load could have been when interviewee GA2 attempted to register an account unsuccessfully in the beginning, because there already was an account with the same company, as already mentioned in *Training, Support & Documentation* and *Learning*.

On one occasion interviewee GA2 thought that she had set up a pipeline rule and asked us why the it did not fail the pipeline. We forwarded this question to Debricked, whom concluded the following:

- She had set up the condition "*If dependency is vulnerable and CVSS **is** high and debAI is at least 70*".
- "*CVSS **is** high*" meant that it should be exactly "*high*" and not include more sever severities ("*critical*").
- for the CVSS condition she probably meant the rule "*CVSS **is at least** high*".

Our contact person at Debricked also noticed that "*high*" was the first dropdown option when setting up a rule instead of "*at least high*". He said this was a user mistake that could be very easy to miss. They decided to change the dropdown menu order in Debricked, so that "*at least high*" was the top first alternative when setting up a pipeline rule condition. In addition, he mentioned that it would probably be more intuitive to have it like this, and that it would help reduce such user mistakes in the future.

Many of the issues mentioned above regards issues that affected interviewee GA1 and interviewee GA2 during the learning process. We would like to clarify that *after learning* both interviewees thought that Debricked were simple and easy to use and that it covered their needs. They both stated that they understood how Debricked worked now and did not think any part of Debricked was hard to interact with any longer. However as they also mentioned Debricked's interface being unintuitive to use (but not necessarily *difficult) any longer*, it seemed to affect *User Satisfaction* negatively.

**Robustness, Stability & Reliability**. Previously we described in *Learning* and *Comprehension* that interviewee GA1 had several vulnerabilities identified by Debricked for the same package. This could be perceived as Debricked being robust, that it rather would detect one vulnerability too many than too little which seems best from a security and accuracy point of view.

None of the interviewees GA1 and GA2 mentioned any false positives. Both interviewees thought that Debricked's suggestions were good and optimal. Interviewee GA1 thought that for the duration that they have been able to use Debricked, they would say that the tool is reliable and accurate. They thought Debricked worked like it should and that there was no strange or inconsistent behaviour. Interviewee GA2 thought for its purpose of detecting dependency vulnerabilities that Debricked could completely do the code analysis itself. It sounded like both interviewees trusted Debricked as a product. The interviewees stated that they did have some trust into the vulnerability scores that Debricked used even though they were not completely aware of how they were calculated. However interviewee GA2 stated that the vulnerability scores seemed reasonable of what she understood about them i.e. she put trust in some extent in them.

Neither interviewee had encountered any delays in Debricked. However, interviewee GA2 stated that regarding time in general using Debricked had not saved them any time, as Debricked added an additional work task, however it had saved them worrying. She said that Debricked was a simple way to get that extra security check on their project, with information that she knows most developers do not put any significant amount of time and effort into, she added. In addition, interviewee GA1 thought that Debricked had improved their work life as they otherwise would not have found and fixed the vulnerabilities that it detected. When Debricked alerts about vulnerabilities it could be an eye-opener that it is time to look over the packages and update them, interviewee GA2 added.

Interviewee GA2 explained that the vulnerable packages often tended to be old. That Debricked identified vulnerabilities for old packages, which often made sense to the interviewees, probably increased the trust or assumed reliability and accuracy of Debricked. Errors or issues that the interviewees encountered likely affected the interviewees' impression of Debricked negatively, however they still seemed to perceive Debricked as reliable. For possible improvements to Debricked interviewee GA1 mentioned automatic generation of pull-requests which could solve vulnerabilities, this was an upcoming feature they knew that Debricked was working on however. This feature is further described and was tried out in project *M*.

**Ease of Remembering**. Project GA was the only project that the *Ease of Remembering* factor would be applicable to, even if it was just by a small degree. However, both interviewees thought that if they would not use Debricked for 1-2 months, that it would still be easy to remember how to use Debricked.

**User Satisfaction**. Between 0 to 100, interviewee GA2 would rate Debricked 80 with respect to user satisfaction. "*With a little minus for the user interface and possibly for the [negative] understanding of how to use it intuitively*", she added. But it covered their needs, so she gave it a high rating of 80, she continued. Interviewee GA1 could not decide completely but would give it either 75 or 80 on the same grounds as interviewee GA2. Both of the interviewees said that they would absolutely keep using the tool. Interviewee GA2 mentioned that Debricked was an advantage to have and that there was no cost attached to just keep using it either, as it is free. She pointed out that as long as Debricked is free, it would be a no-brainer. If it would begin to cost they would have to weigh what it is worth and compare it to its cost. Interviewee GA1 agreed. If interviewee GA2 had time she would try to add Debricked to their primary pipeline, too, she commented. Both interviewees would recommend it to their colleagues. Interviewee GA2's reason for recommending it was that it was a good tool and as soon as a user understands how to use it, it would become very simple. It adds an extra security check on the source code and helps developers with dependencies which she knew most developers did not put a lot of time on, she added. Interviewee GA1's reason of why was that Debricked saved worrying with just a little more work for them. Interviewee GA2 concluded that it was very nice and useful to get to try Debricked, and interviewee GA1 nodded. Debricked fit well to its purpose, it did what it was supposed and meant to do, and it was easy to add to their workflow, interviewee GA2 summarised.

### 5.1.2 Project CB

Project CB has one interviewee, namely interviewee CB1. His work task was *code review* (see Table 3.5) and he evaluated Debricked solely.

From the introduction survey, interviewee CB1 was asked how he expected an AI-based code review tool to affect their work, e.g. regarding quality and productivity, of which his response was *"I expect it to take care of the simpler issues so that I can focus on verifying that it has the correct behavior instead and finding issues on a higher level."*

**Training, Support & Documentation**. Interviewee CB1 had heard previously about Debricked, but had not used it before. However, he knew that there were other similar tools. Interviewee CB1 found our introduction training to be sufficient for getting started. However, he said that he had not had time to check the documentation. He stated that he would have wanted Debricked to provide an introduction about the diagrams and the scoring values i.e. how critical they are practically, and what they are based upon.

While setting up the tool for the project, we observed that there was no documentation about how to use the source-codeless feature for BitBucket with Gradle, even though it existed for GitHub and GitLab (with Gradle or Maven). We had to receive support from Debricked through our contact person by email. We received response in a timely manner with a simple solution, further described in *System Interplay*.

**Learning**. Interviewee CB1 mentioned that it was obvious from the beginning how the tool was supposed to help him. He pointed out that his first impression of Debricked was that it was very simple to understand. Still, he stated that initially the interface of Debricked was a little troublesome to navigate.

**Work Process Interplay**. Interviewee CB1 only used Debricked occasionally and not as part of his regular workflow, i.e. he did not integrate Debricked in his workflow. When interviewee CB1 started using Debricked he discovered it reported very few vulnerabilities found. He had not had time to configure the pipeline behaviour, and did not feel a need to do so since there were so few vulnerabilities. Interviewee CB1 mentioned that after learning Debricked, there was no problem navigating it. Interviewee CB1 stressed that a feature he thought was missing from the interface of Debricked was the ability to identify the parent dependency of a vulnerable dependency.

Interviewee CB1 mentioned that sometimes the vulnerability scan would run in the pipeline and complete like no vulnerabilities were found, but that then a result with actual vulnerabilities detected would be emailed to him around 30-60 minutes later. He assumed that the actual scanning did not start until later. Interviewee CB1 likely felt his work flow with Debricked was interrupted because of this.

The interviewee described that their code review process was similar to that of project GA, but with significantly less documented coding guidelines. Their aim is to just generally follow the layered architecture. Within this project, the code review process is taught through informal communication rather than through formal training by the most experienced developer(s) actively developing project CB.

**System Interplay**. Interviewee CB1 had a few problems when attempting to integrate Debricked into the pipeline of project CB. To make Debricked run source-codeless, a step prior to the vulnerability scan had to be added. Also, interviewee CB1 told us that project CB had a different setup than the default configuration found in the documentation. This might be the reason interviewee CB1 contacted us for support in integrating Debricked source-codeless. We estimate it took interviewee CB1 45 minutes of troubleshooting before contacting us, after which we helped him an additional 45 minutes to get Debricked to work source-codeless in the pipeline. Interviewee CB1 thought that if they would have used the default pipeline setup compared to the documentation, there would have been no problems getting Debricked integrated.

In addition, interviewee CB1 thought that a minor annoyance was that every time he entered `debricked.com`, even if just closing and re-opening it, he would have to click *"Log in"* to see the overview of project CB. That was, because when clicking *"Log in"* a user is being redirected to `app.debricked.com`. Upon learning about the upcoming automatic generation of pull requests feature, he thought it would make using Debricked smoother, but suspected it would not work for dependencies to dependencies.

**User Control**. Interviewee CB1 perceived himself to have less than desired control, because of the previously mentioned issue in *Work Process Interplay* where a vulnerability scan with no vulnerabilities found later were followed up with an email that vulnerabilities actually were found, with the two results being inconsistent with each other. He pondered if it had anything to do with the pipeline configuration, but it seemed to be outside his control.

**Comprehension**. As previously mentioned in *Learning*, interviewee CB1's first impression was that Debricked was easy to understand. Initially the interface was not intuitive, but after using Debricked he had no problems whatsoever, he remarked. However interviewee CB1 mentioned that he did not understand the diagrams, but suspected he would understand these if he read up on them. He also stated that he did not understand how a vulnerability received a certain score, which may have been caused by the AI-based nature of the tool.

Interviewee CB1 thought that some of the solutions suggested by the tool were hard to interpret. For one of the vulnerabilities, he did not understand the suggested solution at all, or interpreted it as there being no solution. In another vulnerability, he thought the suggested solution was to just remove the dependency completely, which he stated made no sense at all. Both of these cases regarded indirect dependencies he had to investigate, interviewee CB1 reported, which most likely did not make it easier for him. Even though the suggested solutions were hard to interpret, interviewee CB1 found Debricked to give him a good indication about what vulnerabilities there were in the project.

**Robustness, Stability & Reliability**. Interviewee CB1 told that it was hard to tell if Debricked was accurate in its suggestions. As already mentioned, interviewee CB1 stated that the tool found much fewer vulnerabilities than he had expected. Interviewee CB1 also explained that one of the vulnerabilities found did not affect project CB since it concerned a database which they solely used for testing. He pointed out that it is about the accuracy, in that how much it finds compared to how much there really is, which is difficult to asses unless someone makes a manual analysis of the dependencies. Interviewee CB1 also mentioned that he would not rely on Debricked for security purposes if he would have a pull request to merge in. If he had a requirement in a project to not have any vulnerabilities in his dependencies, he asserted that he would probably not rely on Debricked at the moment. The previously mentioned issue under *User Control* about the vulnerability scan passing and interviewee CB1 later receiving a report with vulnerabilities later made him view Debricked as less reliable as he otherwise would. Regardless, interviewee CB1 said he thought it was very useful information, but that the accuracy was hard to tell unless someone performed an analysis of the dependencies.

Interviewee CB1 had not encountered any errors, but mentioned Debricked giving him an invalid link to a file of project CB in BitBucket. We attempted to verify it and also found an invalid link when attempting to open a file (prompted from "*Suggested fix*") with a branch different than the base branch. This was not investigated further.

**User Satisfaction**. Between 0 to 100 interviewee CB1 stated that Debricked was difficult to rate as he had not had time to use it. He really would need more time with Debricked before being able to answer, he added. If we noted that he had not had time to use the tool, he would rate it to "*maybe 65*". The major reason which he felt drew the rating down remarkable was that Debricked could have been clearer. He commented that it is still more than the average of 50 so it is still a useful tool. He stated Debricked provided useful information, but that he could not rely on it. He contemplated if he would keep using Debricked after the case study, and stated that he would let it be running in project CB for a while at least, and afterwards see what happens and decide. Before even considering recommending his colleagues to use Debricked interviewee CB1 mentioned that he would first like to use it a couple of months.

## 5.2 Repeated Code Analysis

### 5.2.1 Project M

Project M has two interviewees M1 and M2. Their work task was *repeated code analysis* (see Table 3.5). They evaluated both Debricked and DeepCode. Only interviewee M1 was able to take part in the interview. Interviewee M1 had very high expectations before trying out Debricked and DeepCode. He had stated "*When I heard about these tools I had the feeling that this is going to solve every kind of issue. Then after on I realised that 'Ok, they [Debricked] are focused on packages'...*".

Interviewee M1's response to the introduction survey was "*I expect it to help in improving the code quality and productivity*" while M2's reply was "*I expect that it will make it easier for me to notice potential problems with the code, and help me write better and safer code*".

**Training, Support & Documentation**. Interviewee M1 had no previous experience with either Debricked nor DeepCode. Interviewee M1 found our introduction was good because he had been able to ask any questions that came to mind at the time. As for documentation, interviewee M1 mentioned that he did look into some tutorials of how the tools worked. Interviewee M1 found that the documentation for creating a new pipeline in Debricked made the setup "*okay and not hard to implement*". He remarked that since Debricked is quite new there is room for improvement of the documentation. As for support, interviewee M1 said that he had received very useful help from us. In the beginning however, interviewee M1 and M2 told us they had difficulties in knowing how to use Debricked to solve vulnerabilities. We also observed that they seemed to have difficulties, of which they confirmed at that time. Therefore we arranged an online meeting between the participants in project M and project GA, of whom interviewee GA1 explained to interviewees M1 and M2 how they used Debricked and solved vulnerabilities. Interviewee M1 told us one key aspect they learned from that meeting was to look for the purpose of each vulnerable package they find, before handling it.

**Learning**. When starting to learn the tools, interviewee M1 expressed that he did not have any difficulties, they were easy to learn and straight forward to use. Solving the found vulnerabilities on the other hand however could be tricky sometimes, he pointed out. Interviewee M1's first impression of the user friendliness for DeepCode was good, but not as good as for Debricked. Interviewee M1 explained that Debricked was really easy to learn and very user friendly.

**Work Process Interplay**. We observed that the interviewees used Debricked daily, but only used DeepCode the first two days, neither tool in their normal workflow. M1 told us that for Debricked fixes, he created a separate branch from the base branch from where he attempted to solve the vulnerabilities that Debricked detected. When interviewee M1 attempted to add interviewee M2 to the project in the interface of Debricked, except for email he also had to choose the password for interviewee M2 (whom could change it later), which we observed there may be better security procedures for (e.g. that interviewee M2 could receive a link in his email to choose his password himself).

Interviewee M1 said that he focused on the most critical vulnerabilities that Debricked reported. Interviewee M1 told that he used the results from DeepCode more as a reflection of what vulnerabilities there are in project M, but technically did not solve them. He told that he saw the DeepCode suggestions as guidelines about what issues there are in project M and how other developers had fixed the same issues.

We observed that interviewee M1 at first attempted to fix the vulnerabilities Debricked detected by manually editing the package configuration file, according to the suggested fix diff by Debricked (which he pointed out also seemed to be on the wrong line number). As that did not work (we deem the file likely became corrupted), he sought help from us, and we told him that he was supposed to update the vulnerable packages through the package manager that was used in project M. We also told him that another option was to try out the new automatic pull request generation feature that Debricked had just rolled out a week earlier. He went for the latter, trying out the new feature. Unfortunately, the solved vulnerabilities in the end of the case study could not be merged in to production, due to project M being in its end phase. As for the issues detected by DeepCode, he did not attempt to solve them but deemed that DeepCode's detected issues should be possible to solve as well.

**System Interplay**. Interviewee M1 said that Debricked was easily integrated with project M because in project M they did not have any previous pipeline: It was simply just to create a new pipeline with the default configuration from Debricked documentation. M1 stated "*...a normal programmer doesn't have to put a lot of effort into understanding how it works. It is simple and straight forward, like, you just set up your repository and specific branch, and once the pipeline is created you get a reflection in Debricked, and of course, it will perform some checks and give you some results...*".

He appreciated the BitBucket pipeline and GitHub integration to Debricked. The new automatic pull request feature only worked for GitHub, which interviewee M1 disliked and which made it harder for them, he told us. He explained further that Debricked would be more useful to the case company if this feature was available to BitBucket. Henceforth, Project M was cloned to GitHub to try out this feature. Nothing stopped interviewee M1 from pushing the changes made on GitHub to BitBucket as well, which he did, he said, meaning this could be seen as a temporary work-around until this feature is supported on Debricked's BitBucket integration as well.

Interviewee M1 also mentioned that if you want to integrate these tools and put them to better use for an upcoming project, that you need to plan accordingly. He meant that for projects with not-so-great overall quality like project M, the vulnerability fixes might not integrate well for the project. He gave project GA as an example of a project where Debricked could be really helpful as it is a well maintained project that has went through several restructurings.

During programming, interviewee M1 used extensions to his IDE and editor, like e.g. linting extensions. Testing was performed manually in project M. That is why the developers in project M would have to repeat all of the testing if all suggested changes from Debricked were integrated. Interviewee M1 further explained that they currently did not have the time and effort needed to repeat all this testing in project M.

**User Control**. Interviewee M1 thought that the tools provided enough control to the user. For Debricked, he liked the possibility to work in a team. What he meant was that in Debricked you can add people to your project, to collaborate on solving the vulnerabilities for the same project. Also, there is also a feature to comment on the found vulnerabilities. Interviewee M1 appreciated the possibility to use Debricked however you found suitable. He further explained that you can solely use the vulnerabilities found by Debricked as a "good to know" once in a while, or you can use Debricked in the pipeline however you found suitable. In the pipeline he explained that there is also a possibility to configure when and how Debricked should run.

One thing interviewee M1 remarked though, is that there is no control over which part of the project to scan, as e.g. project M is split into both frontend and backend in the same repository. Currently, you can only scan the full project. Previously we mentioned in *System Interplay* that interviewee M1 mentioned that if you want to use these tools for an upcoming project, that you need to plan accordingly. We identify the absence of being able to scan part of the project as one factor in this, as splitting frontend and backend to separate repositories is one way to achieve separate scanning. Also previously mentioned is that there is no restrictions for e.g. pushing vulnerability fixes done using the GitHub integration to the same project on BitBucket, which could be seen as form of user control.

As interviewee M1 tried out the pull request generation feature by Debricked, he noticed that there were no option to stop, pause or cancel the generation, once it has begun. The reason for cancelling it, would be if the process took too long time or seemed stuck.

**Comprehension**. Interviewee M1 thought that both Debricked and DeepCode were intuitive to use in their interface, and that they were helpful and user friendly. M1 thought that these tools could provide really good facility for developers as they are able to improve code quality and overall performance. For Debricked interviewee M1 pointed out that Debricked was simple and straight forward to use as well, and that there were not many things you needed to keep in mind or consider while using it. He remarked that he did not need to think hard about what the different things meant, and that the list of vulnerabilities were sorted by severity by default. Interviewee M1 appreciated how Debricked very clearly showed the severity of the vulnerabilities with different colours. To clarify the reader the severity of critical vulnerabilities are red, while it is orange for high severity and blue for medium severity. How the severities were displayed allowed him to quickly could get a comprehensible overview of the vulnerabilities, interviewee M1 thought. For DeepCode he pointed out that its layout could be improved, referring to the layout of Debricked being a good example that could be used as guidelines. He thought that Debricked was more helpful in a simple way than DeepCode, interviewee M1 said. He further explained that DeepCode had provided suggestions which did not 100% reflect the issues in project M but were very similar, and that the suggestions depended a lot more on context (we and DeepCode according to their website interpret such incorrect suggestions as false positives). This required him to think through how much were similar and to what degree the suggestion is actually solving the specific problem. Interviewee M1 thought he understood why these tools gave the results that they did and about how the tools worked, though.

Interviewee M1 stated that it was really hard to know if some package was going to have a problem with other old packages, and that this was something Debricked could help with. He clarified that when introducing a new package it is good/helpful to know that there are no vulnerabilities introduced with it. Interviewee M1 stated that understanding the suggestions in how to fix a vulnerability in Debricked could be tricky. He was unsure of how to update the packages correctly through the package manager according to the vulnerability fix that Debricked suggested. This was one reason for him trying out the new pull request generation feature.

Interviewee M1 told us that when he tried out Debricked's feature of automatic pull requests, he encountered some long delays without any explanations from Debricked. When clicking the button to generate a pull request for a vulnerability, there was a progress bar and percentage indicating how far the process has come. There was no other feedback except for the percentage number. Interviewee M1 told us that sometimes the generation seemed stuck (as the percentage number was not moving), and that it was hard to tell if the problem was on the side of Debricked or the project. He said that he did not understand why the delay happened, and that he expected an error or explanation so he could avoid such delay in the future, but nothing appeared. There was no way to stop, pause or cancel the pull request generation, which we already mentioned in *User Control*. Interviewee M1 suggested that a possible improvement to Debricked would be that when a pull request generation takes more than a couple of hours, to at least give some feedback and explain why it is taking that much time, and perhaps give the user an option to cancel the generation.

One thing that interviewee M1 had a hard time understanding was when he fixed one vulnerability in Debricked and one or two additional vulnerabilities appeared. *"That was tricky"*, he remarked.

**Robustness, Stability & Reliability**. When interviewee M1 previously mentioned that you need to plan accordingly in order to use these tools, it could be perceived as such tools not yet being able to handle all projects in a ideal way, and therefore not being as stable or robust currently.

Interviewee M1 stated that in general the tools Debricked and DeepCode gave good results and are great tools. He thought that they provide good facility to developers and he had the impression they are emerging and improving step by step. Interviewee M1 thought Debricked was a more powerful development than DeepCode when comparing the user friendliness of the tools.

Regarding reliability and accuracy of DeepCode, interviewee M1 thought that DeepCode could improve the explanations and the fixes suggested themselves. Comparably, he thought that Debricked really could pinpoint both the vulnerability detected and the suggested fix.

Interviewee M1 told that in order to tell if the tools were reliable and trustworthy he would need to experiment more with the tools. He suggested it would have been easier to know if they were trustworthy if he had tried the tools on a project without shortcomings. The structure of project M was outdated, he explained. However interviewee M1 said that Debricked made him not need to worry about if a dependency was safe or not every time a new dependency was added, which he stated were about every second or third work day. However interviewee M1 told us that once he attempted to make the change *by hand* through a text editor in the package file to solve a vulnerability detected by Debricked, however that went unsuccessful and seemed to have corrupted the package file.

Regarding delays, as previously mentioned interviewee M1 said that he had an issue with delays regarding the automatic pull request generation feature by Debricked. The longest automatic pull request generation took 2-3 days, however he did not get any feedback of the estimated time left. He stated that the normal delay for this would be a couple of minutes or a couple or hours, i.e. it varied greatly. He contemplated if the structure of project M had anything to do with the delay, and thought that the not-so-good structure of project M increased the delay. We observed that the generated pull requests fixes for different vulnerabilities often had to be generated sequentially, because otherwise there would be a conflict between the pull requests, i.e. one pull request had be generated after another one was merged in. Without this issue we think the delay described above would not have been such a large issue. Then when it had been merged, interviewee M1 mentioned that there was no noticeable delay until the vulnerability analysis results updated.

Interviewee M1 said that he had not received any errors while using Debricked or DeepCode. Moreover we think that it was the absence of error or feedback for the delay when generating pull requests that negatively impacted his user experience. Interviewee M1 even recommended us to contact Debricked regarding either the delay or absence of feedback, as he elaborated that he had no problem if the problem was on the side of project M or of Debricked.

**User Satisfaction**. Interviewee M1 stated that he would give Debricked a rating between 60 to 70 out of 100, and the reason for why it was not higher was that for Debricked to be really helpful in a project, he said, the project would have to be well planned i.e. maintained and well-structured. He followed up with that it was great that Debricked was being continuously improved so that the issues he had while using it could be fixed. Also, the previously mentioned delay of generation of automatic pull requests with insufficient feedback likely impacted impacted interviewee M1's user satisfaction negatively. For DeepCode he did not give a specific rating, but we estimate it would be around 50. For DeepCode, interviewee M1 thought that it could improve the explanations and the suggestions it gave him, which he pointed out were more generalised and not specific, like Debricked which pinpointed the problem and told him *"Fix this, then that vulnerability will be solved"*. He thought that DeepCode could improve a lot in its way-off suggestions.

When asked if he will keep using any of the tools, interviewee M1 told us that he will definitely discuss it with the team or the person who is in charge (for the next project, as project M is in its end phase) to consider using these tools. He would definitely try to promote using these tools. In his personal projects he did not see any reason to why he would not use these tools. Interviewee M1 would recommend the tools to his colleagues. He was glad he could explain the issues he had in this case study such that the products could improve.

## 5.3 Code Analysis

### 5.3.1 Project CF

Project CF has three interviewees CF1, CF2, and CF3. They performed the work task *code analysis*. They were supposed to evaluate both Debricked and DeepCode, however in the interview it showed that they due to time constraints only were able to evaluate Debricked. Only interviewee CF1 was able to take part in the interview.

Interviewee CF1's response to the introduction survey was *"At the beginning it will slow down the process and I think some "not important" bugs will haunt us. But overall I think the code quality will improve and we developers as well"* while CF2's reply was *"Help me produce better and cleaner code"*. CF3's response was *"I expect it to affect my work positively by hopefully getting help catching things I might miss while reviewing a pull request, or while writing code"*.

**Training, Support & Documentation**. Interviewee CF1 did not have any previous experience of Debricked but thought that its name sounded familiar. As for training, interviewee CF1 thought that our introduction was enough and that there was not anything that was missing. He also thought that the instructions we gave him about setting up Debricked and DeepCode were straight forward. However, interviewee CF1 pointed out that he did not really understand the severity scoring system that Debricked used, about what it is based on and how much value he himself is meant to put on these scoring values when using Debricked. He had no idea about what the abbreviations of the scoring system meant either, but told us that he was able to read up on those in the documentation (on Debricked's website). Interviewee CF1 told us that the second time he logged in to Debricked, he received a tutorial which answered and explained all of his questions. He remarked that if he would have received the tutorial the first time he logged in, that then everything would have been straight forward from the start. At that point however, he had already learned most of Debricked by himself.

**Learning**. Interviewee CF1 told that after the introduction he thought it was quite obvious how Debricked was supposed to help him i.e. for checking vulnerabilities among dependencies. Interviewee CF1 told that when learning to use Debricked, it was very simple to integrate and also to interpret, and that is was mostly the details he had to learn about before being able to understand the interface and interpret the results provided. Details that he did not understand initially were what the scoring values meant and the abbreviations that Debricked used in their interface.

He remarked that after the tutorial however, that all of Debricked suddenly were pretty simple to him. The tutorial had answered all his questions. In contrast, CF1 told us that something they got stuck on in the beginning was that it was very unclear what role each dependency had in the project. This was not something Debricked helped them with at all, according to CF1, and they ended up investigating the role of the project dependencies as a group. He told us that it was quite the *"detective work"*, which afterwards facilitated for them when looking at the vulnerable dependencies. Something that would have facilitated a lot, interviewee CF1 mentions, was that when looking at a vulnerable dependency, that Debricked also says which project dependencies that actually are using this vulnerable dependency. What they had to do at the time was to look in the package dependency file to find all the places where a vulnerable dependency was used, according to interviewee CF1. As a side note interviewee CF1 mentioned that because of the lacking dependency tree information in Debricked, through all of their work, that he and the other people in project

CF had learned a lot about the dependency tree and how it really looks like.

**Work Process Interplay**. Interviewee CF1 stated that the only actual tool that he and his team had time to use was Debricked. They had not had time to try out DeepCode, he mentioned. We observed that also for this project CF1 had to choose their passwords when adding his team members in Debricked. For Debricked he told us that they had integrated it into their BitBucket pipeline and afterwards checked the vulnerability scan result on `debricked.com`. He informed us that there were a lot of vulnerabilities detected, for which he also typically noted the vulnerabilities often regarded outdated packages. Afterwards, interviewee CF1 told us that he and his frontend team, including the project leader, discussed one whole afternoon about the results. They decided that the cost of time and effort to fix all of the vulnerabilities were too high, according to CF1. However, they still kept Debricked integrated after the analysis, such that the team get updated reports from Debricked, just for them to keep in mind in case they get leftover time, according to interviewee CF1. When briefed about the possibility of automatic generation of pull requests, he responded that this was not something that could have worked in project CF as it would likely break the project. He further explained that one of the major requirements of project CF were being backwards compatible with IE11 (Internet Explorer 11), and package updates might likely break that requirement, and merging them to the base branch would require a lot of time testing. In addition, interviewee CF1 pointed out that they can not test the project completely before it is deployed because they do not have access to the actual production environment. Interviewee CF1 also noted that most of the vulnerabilities identified had to do about an old version of react, meaning that in this case they would need to update their whole framework. IE11 makes it extra sensitive, according to interviewee CF1, and therefore automatic pull requests would probably not work for them.

**System Interplay**. Interviewee CF1 had integrated Debricked through the BitBucket pipeline, and according to him it was very simple to integrate their project with Debricked.

**User Control**. Interviewee CF1 had during the interview pointed out that Debricked should help him in his work, but not control it. Other than that, he did not mention any aspects about the user control in Debricked (of what he had been able to try out in Debricked). Perhaps more feedback would have been given this factor, if more than a one-time code analysis had been performed.

**Comprehension**. According to interviewee CF1, it was easy to integrate the project with Debricked. In addition, he did not have any problems with the interface anywhere. Things that CF1 did not comprehend completely or thought was unclear though, were mostly related to the learning step (see *Learning* factor). After learning to use Debricked however (after the second time he logged in that he received a tutorial, once again see *Learning* factor), he thought that Debricked as a whole seemed simple to understand. Still, after learning to use Debricked, interviewee CF1 seemed hesitant and unsure about how much weight to put into the scoring values that Debricked used and what it was based upon. Examples interviewee CF1 provided were how he should treat a score of 9.5 compared to a score of 9.7, or that he knew that 9 was a sever score but did not know what decided the exact score of 4.5 to another vulnerability (and how he should treat it). Was a score of 9 something that he should stop everything he was doing immediately to fix, or is it just a "*I will fix it when I get the chance*", he asked us. This seemed important, as he followed up that question with the statement "*Somewhere in the way I need to have a clear mind as a developer and think through if it is really as such a big problem in the whole picture at the moment, too, or if it is just a 'good to know, I will fix it when I get the chance'*".

We deem that if Debricked would have showed what a vulnerable dependency was used by in the project, and also what role that dependency had, it would likely lead to improved comprehension and less work time to achieve the same results with Debricked, especially for new users to Debricked. It would also probably facilitate for current developers using Debricked, not needing to manually look up this information themselves. Related to this, something that interviewee CF1 stated that he thought was great, was when Debricked showed examples of how a vulnerability could be exploited in code. In that way, he said, he could see how the vulnerable package could be used for malicious actions (what they want to protect against when using Debricked). This seemed to improve comprehension. Interviewee CF1 also stated that for Debricked, he thought they had a clear purpose.

**Robustness, Stability & Reliability**. In contrast however, as previously mentioned interviewee CF1 had noticed that a lot of vulnerabilities had to do with outdated packages. To reliability (and some comprehension) this is related in that it seemed to have made much sense to him that outdated packages typically are more prone to have known vulnerabilities. This likely affected interviewee CF1's perceived reliability in that it agreed with what he already knew about old packages being prone to vulnerabilities, such that he could put further trust into the scores.

Interviewee CF1 himself mentioned that he did put some trusts into the scores that Debricked assigned the vulnerabilities, with the reason that Debricked seemed like a serious company who were serious about their business. He assumed, that Debricked knew what they were doing, in that e.g. if Debricked showed something as a *"9 out of 10 severity"*, then he assumed that it really was a critical vulnerability.

**User Satisfaction**. Between 0 and 100, regarding only the security vulnerabilities, interviewee CF1 would give Debricked *"definitely 100"*, while he would give Debricked as a whole a rating of 90. He mentioned that what would draw down the rating from 100 to 90 was the user friendliness. He further explained that it had not anything to do with his experience with the tutorial as that just could have been a coincidence, but more the specific missing feature that Debricked does not show what a vulnerable dependency is used by, i.e. what answers the question *"what is the direct (parent) project dependency that uses this vulnerable dependency?"*.

Well agreeing with the high rating, interviewee CF1 thought it was a very good tool, it gave him the vulnerabilities there were and he has his own decision to take in how much the vulnerabilities matters in the big whole project, because that is not something Debricked probably does, instead it looks at the vulnerabilities individually and makes an individual assessment, according to interviewee CF1. In addition, he commented that *"Debricked has a clear purpose which it fulfills"*.

Interviewee CF1 would argue for keeping Debricked in project CF after the case study. He had not had the discussion with his project team yet whether they will keep Debricked integrated or not, but he would recommend to keep it in a in a way that they have a to-do list of security vulnerabilities, he told us. *"I would recommend my colleagues to definitely test it. I cannot say that a project stands and falls on it, but would tell them to definitely try it out and evaluate it themselves"*.

At the moment Debricked was free, but if it would begin to cost interviewee CF1 mentioned that his answer instead would be *"no"* and to not keep it. Interviewee CF1 further elaborated that project *CF* was a fixed-price project where they tried to keep the costs low, and that he could not motivate to keep Debricked in case it had any cost linked to it. *"It is a choice we are aware of to run with packets that are older, but we need to consider that it should work* *in IE11, which is priority"*, he mentioned.

### 5.3.2 Project H

Project H has three interviewees H1, H2, and H3. Their work task was *code analysis* and they evaluated both Debricked and DeepCode. Only interviewee H1 was able to participate in the interview.

Interviewee H1's response to the introduction survey was "*I expect it to keep the same or increase the quality depending on how well it performs the code review. I expect my productivity to increase since I won't have to spend my time on other tasks*" while H2's reply was "*I don't think it would affect anything in particularly. I kinda see it like another reviewer but instead of a real person it's an AI/bot that does the code review. It may affect projects more where there's no reviewers, i.e. only one developer*". H3's response was "*It will make the code review part faster which makes it possible to pump out more code. However I feel that if you do not do the code reviews you might not learn so much from eachother*".

**Training, Support & Documentation**. About previous experience, interviewee H1 mentioned that he had heard about the "*buzz word discussion about how 'AI will take over your code review'*" but that it was new to him and he had never seen it for real.

Interviewee H1 was not able to attend our introduction but he read and performed the instructions we gave him for logging in to Debricked and DeepCode to view the results for project *H* that we had cloned to GitHub previously. Even though he did not attend the introduction he still thought that he could navigate around Debricked and DeepCode and interpret their results correctly. Interviewee H1 had not had to use the documentation to look at the results that Debricked and DeepCode generated, but thought that he would have to look through it thoroughly if he had to set up and connect Debricked and DeepCode to the cloned GitHub repository himself (what we had done previously). He did not need any support outside of the already-included instructions mentioned above. However, when following the instructions another interviewee H2 emailed us about help in logging in to DeepCode; He had encountered the same DeepCode redirect problem as already encountered and described in project *CB*. Before we had time to reply, he emailed us that he did not know if it was a temporary issue or not but that he had gotten it to work just soon after.

**Learning**. Even though interviewee H1 had no prior experience with these tools he still told us that it was obvious since the beginning how the tools (Debricked and DeepCode) were supposed to help him. he also thought that he could navigate and interpret the problems correctly since the beginning. He thought that it would have been harder to learn if they had to set it up themselves, then they also likely would have had to use the documentation, as already mentioned in *Training, Support & Documentation*.

**Work Process Interplay**. The people at project *H* did not use Debricked or DeepCode in their workflow as this was a one-time code analysis. They seemingly followed the instructions, navigated around the interfaces and interpreted the results of the tools. It was fully optional and up to the team themselves if they wanted to solve any potential vulnerabilities detected, critical or not.

**System Interplay**. DeepCode did not integrate well with BitBucket. As their source code was hosted on BitBucket we deemed that the easiest way to set up and connect the tools (Debricked and DeepCode) was to clone it to GitHub. If we only would have used Debricked we might have integrated it to their BitBucket pipeline, but if we were to use DeepCode with

BitBucket we would have to share the whole sensitive BitBucket workspace (containing other sensitive company projects) with them. In addition as DeepCode was not available as an official app (probably because the app was new or under development), if we wanted to integrate DeepCode to the whole BitBucket workspace we would have to change the workspace settings (for the whole BitBucket workspace) to enable development mode i.e. to allow non-official apps in BitBucket. An action this big, affecting all repositories in the workspace, was unreasonable, according to the people responsible for the projects and the workspace. Cloning the source code to GitHub and sending the people at project *H* instructions for accessing the tools connected to this GitHub repository seemed like the simplest solution.

We also observed that when integrating DeepCode with an online source code repository service such as GitHub or BitBucket, it seemed to automatically start to analyse all repositories as a default as soon as possible after connecting. We did not want DeepCode to get access to all of the repositories connected to our account, and this was possible through the settings in GitHub. For an extra layer of protection, a separate GitHub account was used for this case study.

As we set up and configured Debricked and DeepCode with a copy of the source code of project *H* at GitHub, the perceived system interplay by any member of this project are outside of the setup and configuration process. Interviewee H1 himself remarked that it seemed to be a smooth integration between Debricked and Github, as well as between DeepCode and GitHub. He remarked that it just seemed to integrate so well and that they themselves did not have to think about any server processing or storage, leaving no work left for them (outside the main work process of analysing the suggestions from the results and eventually solving the vulnerabilities). This was the core idea, to leave any configuration and setup to us, as they had limited work hours dedicated to participate in this case study.

**User Control**. As Debricked and DeepCode already were set up with the source code in GitHub by us, the user control for the case study participants of project *H* excluded any setup options. For Debricked usage interviewee H1 appreciated the option of being able to collaborate in Debricked i.e. the possibility to add team members that can comment on vulnerabilities. This could help people to not look at the same vulnerability at the same time, meaning it would not become a redundant "*double job*", according to interviewee H1. In the interview with interviewee H1, nothing more regarding user control was mentioned.

**Comprehension**. Regarding comprehension, as previously mentioned even if interviewee H1 was not able to attend our introduction (the training) he mentioned that he thought he could navigate the interface and interpret the results of Debricked and DeepCode correctly. He thought that Debricked and DeepCode clearly presented the problems they had found. DeepCode had a pretty self explanatory name, interviewee H1 thought. He further explained that "*by its name, 'DeepCode', it was reasonable to guess that it will go deep in the code*". Interviewee H1 thought that he had an overall idea of how DeepCode worked, and how it went through the lines of code to check for issues or improvements to be made. He mentioned that DeepCode found one false positive but at the same time he realised why DeepCode had found it, which was because it was an "*ugly solution*" in the GUI, according to him. It had to do with a boolean `isLoading` that is set to **true** in the beginning of the function, and set to **false** in the end of the function, without visible being used anywhere inside the function (but that worked as intended showing a loading animation in the user interface), according to interviewee H1. DeepCode detected this as an unused variable, which made sense to him, and which also made sense to us when he told us about it.

What he really liked about DeepCode was that for a certain issue that it detected, it actually also showed examples of how you could fix it by showing how other developers had fixed a similar problem. He thought that this feature was really cool, in that it could also provide a more or less 'fix' to a problem. These examples of how other developers had solved a similar problem likely helped him understand the problem better as well as to grasp the actual solution better.

For Debricked however he thought that it more or less performed a dependency check, and mentioned that theoretically you could do the work that Debricked did by going through the NPM pages and *node_modules*, what they did with the help of AI. However knowing how Debricked works, in hindsight we knew that to be a false statement in that these security vulnerabilities are found by Debricked by scanning the web: vulnerability databases, GitHub issues, websites, forums, etc. He compared Debricked to be very similar to the `audit` command for another tool called *npm* (which does not use any AI), but stated that Debricked gave an overview of the vulnerabilities in a smoother and nicer way than what *npm audit* does. We deem his understanding of how Debricked worked would have been closer to the actual product if he had read the documentation about Debricked and how it works.

Regarding comprehension, one of the vulnerabilities that Debricked detected was about a package which was discontinued, according to interviewee H1. He said that he did not really know how to continue to solve that vulnerability. Having discontinued packages that are vulnerable may take a lot of effort to find a substitute to. We deem this to likely be an issue because of the project planning or maintenance, at least regarding packages who by our impression in the case study so far do not seem to be prioritised.

**Robustness, Stability & Reliability**. Regarding accuracy interviewee H1 thought that for project H, DeepCode seemed to have worked not great but not too bad either, it did not really find any critical problems in the project. He thought that DeepCode would probably have worked better and been more useful for code reviews in projects with larger code bases that used statically typed programming languages like Java or similar, or just large code bases in general. Interestingly enough, interviewee H1 thought that this lack of issues detected in the project by DeepCode had to do about how much DeepCode could detect, while another reason *could be* that project *H* consists of high quality code with few issues. Compare this with the survey results for project *H* in Table 5.1, where the quality was rated as *excellent*, *fair*, and *good*, respectively.

When asked if he thought the tools were reliably or accurate, interviewee H1 responded that "*Debricked gave us a list in a page where they showed how they [the vulnerabilities] had gotten their points, or at least what points that came from how the AI thought. So that I still think is reliable from Debricked's side*". For DeepCode, he shook his head, and told us that it was unfair that DeepCode gave them a false positive, but if it could succeed in finding additional issues later on, he told us that this could be further discussed.

Regarding issues, according to interviewee H1 most of the dependency vulnerabilities found by Debricked were vulnerabilities that he could not fix (within reasonable time) because they were mostly old or discontinued. He did not know how to fix these vulnerabilities. For DeepCode, except for the occurred false positive already mentioned, he also had an issue with an error when logging in to DeepCode, that the redirect did not work after clicking on the login button. We observed that in the interview the same redirect issue occurred when he was trying to tell or show us something regarding Debricked during the interview, after which we told him a work-around to it. This issue likely made interviewee H1 lower his perception of how robust DeepCode was. This same issue was also already described by interviewee CB1 in project *CB*. However, interviewee H1 did not mention any problems regarding delays.

**User Satisfaction**. Overall interviewee H1 stated that he liked the tools and they seemed well-made, and told us that he appreciated that the interfaces were very good-looking, with nice colours and that they visualised issues in a good way.

Between 0 and 100 interviewee H1 would rate Debricked 90, with the reason that Debricked had a well structured interface, easy integration with GitHub and that afterwards it seemed to do everything in the background by itself, so that he would absolutely give it a 90. Then because of the dependency issues, which he said that you could run *npm audit* for too but with a not as pretty result (he did not seem to know the difference between Debricked and *npm audit* outside that they were different tools in the same category).

For DeepCode interviewee H1 at first said that because of the redirect issue mentioned previously, DeepCode would receive the rating 90, but that he would draw down the rating even further to 60 because that for this project, DeepCode did not show any real strength, but that there is great potential. Even though he gave DeepCode a lower rating than Debricked, he still stated "*DeepCode was the best tool, if I would compare it to Debricked*" (as they are quite different, perhaps it was the most interesting one to use).

When questioned about if he would keep using the tools, he replied "*Maybe for a pet project, but I will not push the company to use AI*". He further elaborated that for his open source projects or similar he might just use these tools because if he is already working alone anyway, it might be a good idea to have someone more check the code, he added. Even if that someone is AI, he implied. He concluded by saying that at least now he knows that these tools exist and that they are very easy to integrate with.

Interviewee H1 mentioned that in the end, they did not end up using the results from the tools to fix any issue or vulnerability. The dependency vulnerabilities that Debricked detected they could not fix and the issue DeepCode complained about was a false positive. He further explained that if someone 'just' gets access to some address, asset or something that they only use for testing, the vulnerability in context would not be *that* dangerous. What he deemed would be more dangerous was if someone got access to backend, firebase credentials or similar, according to interviewee H1. He added that they would have cared a lot more about the vulnerabilities if the project had very security heave needs or requirements, but for project *H* they do not.

### 5.3.3 Project T

Project T has two interviewees T1 and T2 and their work task was *code analysis*. They evaluated all tools being Debricked, DeepCode and Amazon CodeGuru Reviewer. Interviewee T1 was the only interviewee able to take part in the interview.

From the introduction survey, interviewees T1 and T2 were asked how they expected an AI-based code review tool to affect their work, e.g. regarding quality and productivity. Interviewee T1's response was "*Speed up the 'highlighting' process - I expect that it will catch all the common issues that might occur. It might find something that can be overlooked in more complicated code pieces - this I do not expect but rather hope for*" while T2's reply was "*Expect to increase quality and perhaps productivity (questionable)*".

**Training, Support & Documentation**. Interviewee T1 had no prior experience with any of Debricked, DeepCode or Amazon CodeGuru or such tools that use AI, he had heard about some of them, though, he mentioned. He stated he had knowledge and experience of static code analysis tools without AI, that otherwise fit in the same category as DeepCode, Amazon CodeGuru and Debricked. Interviewee T1 thought that our introduction and instructions for checking the results from Debricked, DeepCode, and Amazon CodeGuru were sufficient. He did not mention any use of documentation nor did he ask for additional support.

**Learning**. Interviewee T1 thought that it was very easy to get started with Debricked. His first impression was that the tools worked smooth right off the bat, which was immediately a plus, he pointed out. He had no problems at all when looking at the results from Debricked, DeepCode or Amazon CodeGuru for the first time.

**Work Process Interplay**. Before the case study interviewee T1 had intentionally planted a few bugs and even one multiple-step bug in the project source code, that he knew the static code analysis tools that they already used would detect. For further details about the results, we refer to *Robustness, Stability & Reliability*. Interviewee T1 did reportedly not use Debricked, DeepCode or Amazon CodeGuru in their workflow but included a one-time work session for interpreting the results. He followed the instructions and viewed the results that the tools gave him, and it was optional to fix the issues or vulnerabilities detected by the tools.

**System Interplay**. Due to hassle using DeepCode in combination with BitBucket (as previously described in *System Interplay* of project *H*), and that Amazon CodeGuru seemed to integrate well with GitHub as well as Debricked and DeepCode did, we decided to create a repository there with the source code of project *T* (which we had received by interviewee T1 as a zip file instead of getting GitLab access). We integrated the tools with the GitHub repository and sent instructions for accessing the results for each tool. This setup was very similar to the setup for project *H*, except for the additional tool Amazon CodeGuru used here.

As we set up and configured the tools with a copy of the source code of project *T* at GitHub, the perceived system interplay by any member of project *T* are outside of the setup and configuration process.

Interviewee T1 himself thought the integration was very easy and smooth for all of the tools (Debricked, DeepCode and Amazon CodeGuru). Interviewee T1 stated that there were already a lot of actors in the same space as these tools, some of which he already were using, except that the tools he used and knew about did not use any AI. Throughout the interview he made a lot of comparisons and drew parallels between the tools *he* had used and between the AI-based tools he had tried in this case study, which we will mention where it is relevant in the following text of project *T*.

Regarding system interplay, he added that there is a lot of value to be able to integrate and use different tools seamlessly depending on each project's need, which he mentioned even could have been a whole startup (company) idea in itself.

Some tools that interviewee T1 stated he used along with his describing words about them, were:

- SonarCloud as a static code analysis tool (compare with DeepCode and Amazon CodeGuru),
- Structure101 which surveys the project and helps keep cohesion low (so that a developer do not do something *"stupid"* or use something in the wrong place in the code) (possibly compare with DeepCode and Amazon CodeGuru),

- Local development and linting tools during development locally,
- Tool with forgotten name during interview, which generated a report of the licences used (compare partially with Debricked),
- A dependency plugin/tool by OWASP, used to check for vulnerabilities just like Debricked but without AI, and
- Another tool by OWASP which performed a code analysis just like DeepCode and Amazon CodeGuru.

Interviewee T1 had previously thought it would be exciting to take part in our case study. He told us that afterwards he would begin checking if they would have more use of integrating such tools that use AI in their own work flow. Their tools already covered the most angles, according to interviewee T1, but he said that at the same time it would be fun and surely useful long term with new angles (using AI tools). These AI-based tools probably would fit perfectly in a CI/CD chain, according to him. One interesting difference to the system integration is that DeepCode and Debricked could be automatically set up to analyse new pull requests that occurs in a project and provide their results within the pull request, compared to SonarCloud that looks at the whole code all the time and which does not work on a pull request basis, interviewee T1 pointed out.

**User Control**. Compared to conventional static code analysis tools where someone can read up on and control what rules to trigger, interviewee T1 discussed with us that AI tools does not have such lists of rules, that it is more like *"you get what you see"*. Regarding user control, interviewee T1 thought that there was no control over which rules the AI used, as there typically are no such lists in these AI-based tools that exist for static code analysis tools without AI. Instead of these lists, interviewee T1 came up with the idea that there was a way for users to tune the behaviour of the AI, and that is if they are able to mark false positives. He meant that instead of controlling rules, the AI behaviour could be tuned (through marking false positives). He asked us *"Are some code issues always bad?"* and argued that it depends on context, that the user perhaps should be able to decide whether to mark a false positive as contextual or not.

Something interviewee T1 disliked was that he could not not find or customise (control) the overview to show the issues detected, sorted or separated by each module module. At the moment, the overview showed all of the issues in the source code, sorted by most critical issue to less critical issue, there was no way to get a comprehensive overview of which software modules that are affected without clicking on the issues individually. This issue likely affected comprehension, for which for more details, we refer to *Comprehension*.

Interviewee T1 stated that the AI-based tools he tried in this case study were simple to use and that you get a lot of value if you could not be bothered with the setup of SonarCloud and all its rules. We think he implied that for less effort to integrate AI-based tools the user may have less control but a fully functional out-of-the-box code analysis tool, while if the user wanted to control the exact behaviour and fine-tune the rules and configurations a conventional static code analysis tool would be more suitable. Perhaps a user setting up a code analysis tool nowadays with AI-based tools available too can decide and have the freedom to choose whether he would like great control for more effort with a conventional tool, or little control with less effort with an AI-based tool.

For DeepCode and Amazon CodeGuru there also seemed to be an option to instead of getting issues detected by a model trained on open source code, to train on the source code of the own company repositories. This was not investigated further, but we would like to note that there perhaps are more user control options possible when paying for such service. Interviewee T1 mentioned that having to hand out their source code when using DeepCode or CodeGuru was not positive for project *T*. He meant that there was a greater risk by using these tools of their source code being accessed or used maliciously by unauthorised people. At the same time using SonarCloud and GitHub in project *T* posed a similar risk, but they were perceived as more reliable and trusted than DeepCode and CodeGuru, and as such probably posed a much smaller risk of unauthorised access. He stated that these tools would be more attractive to them if any source code never left them, e.g. if DeepCode and CodeGuru easily could be installed and used locally.

**Comprehension**. Interviewee T1 thought it was obvious how these AI tools were supposed to help him. Conventional static code analysis tools have transparent rules while such AI-based tools seemingly do not, interviewee T1 discussed. For AI, it is more like, to quote interviewee T1, *"you get what you see"*. The AI behaviour could be tuned but that does not necessarily mean that it would get more comprehensive.

As previously mentioned in *User Control* interviewee T1 disliked his inability to customise the overview in that for DeepCode and Amazon CodeGuru interviewee T1 could not get a comprehensive overview of what issues each software module separately were affected by. This forced him to look at all issues as a whole, not necessarily comprehending or realising if there was a certain or a few specific software modules that were problematic according to the tools. Even if the information was summarised about the project overall, they did not tell what parts of the project that had these issues, which he said made the overview look sprawly. It likely decreased the comprehensiveness of the overview which showed all of the issues. Interviewee T1 told us that he had a habit to partly locate where in the code he is, to quickly be able to get an overview. According to him, SonarCloud had the capability for this, where he even could filter for certain rules and where they broke, or all the issues a certain class had, to quickly understand what was happening. He commented that he did not find any way to filter like this in DeepCode or Amazon CodeGuru, and it would surprise him if AI-based tools would be able to filter like that, too, he mentioned. Reasonably, increased comprehension for interviewee T1 would probably be achieved if the issues could be seen for each class or module in the overview. Even if the overview was not as comprehensive, he could still interpret what they were showing, as he had no problems interpreting the results when looking at them for the first time (as previously stated under *Learning*), even if the results were organised in a cumbersome way. He wanted to quickly get an overview however, and did not feel he could get that, he knew what faults there were but not where they were.

Regarding comprehension and simplicity, using AI-based code analysis tools could facilitate the setup process in that you would not need to be bothered by, e.g. what interviewee T1 previously mentioned (in *User Control*) about rules to configure when setting up SonarCloud. AI-based code analysis tools seem to provide less control and possible comprehension about how it works compared to conventional tools, but the simplicity of the out-of-the-box tool would perhaps not require that level of comprehension either, and instead the focus could be laid on the output or other more prioritised things. In other words, we suggest the *required* comprehension for optimal usage may be less with AI-based tools. However, with the option for training on company source code for DeepCode and Amazon CodeGuru however (already mentioned in *User Control*), the required comprehension (level of expertise) for optimal usage may very well be far greater than both previously mentioned alternatives.

Understanding if a detected issue actually is an issue, may be problematic if using an out-of-the-box tool without any transparent rules. Interviewee T1 asked us rhetorically that if for example Amazon CodeGuru or DeepCode were to find an issue for which it tells the developer how he can fix it, but not *why* it is an issue other than the tool itself found it on a lot of other places, what value does the tool provide? "*Should you fix something just because a tool tells you to do so?*" he rhetorically asked us. He did not remember which one, though, but thought that some AI-based tool he tried in this case study actually did seem to give some indication of *why* an issue was wrong. An example we discussed with interviewee T1 was that an AI-based tool could learn unofficial *rules* that are not official fixes in open source communities, but that developers still tend to find and fix. What issues an AI-based tool finds may very well change over time (as the tool improves, or by present best programming practises and habits). Additionally, interviewee T1 pointed out that if an AI-based code analysis tool reported an issue that actually was a false positive, a junior developer might not necessarily know that. However, false positives will always exist, even for conventional static code analysis tools, according to interviewee T1. Interviewee T1 mentioned that SonarCloud easily could summarise each problem and also how test for it, which he added is very valuable for a junior developer. He was hesitant whether these AI-based tools will evolve to be able to offer that. On the other hand however, he mentioned that these AI-based tools may very well discover something new. One thing he mentioned was "*You want to reduce the human factor, and as long as we do programming by hand, there will be human bugs introduced, so I hope that these tools in one way or another, discover something that we have not thought about previously*". We would argue that a code analysis tool has more value to a junior developer than a senior one, as junior developers are more prone to do mistakes. Therefore we deem it crucial for a code analysis tool to have these developers as their target users, especially as e.g. DeepCode and Debricked can provide results on a pull-request basis, compared to SonarCloud that does not have this capability.

Interviewee T1 mentioned that project *T* is a very test-driven project, and that they often test for the cases or bugs that they fix. He explained that the test should fail before the bug is fixed, so that you can actually verify when the bug has been fixed. He compared that with a suggestion from an AI-based code analysis tool which perhaps just states "*this is wrong, and this is how 100000 other developers have done about it*", but is something that does not change the functionality at all (but may be a code smell or bad programming habit), "*Then what is it then that you should verify to test it?*" he asked us. He answered his own question with that there not being a way to test it, and that it becomes more of a "*leap of faith*", to follow the suggestions of an AI-based code analysis tool. *It is very hard to verify the accuracy*, he added.

Interviewee T1 compared using code analysis tools to insurance, that by having a list of transparent rules that he himself had approved, then he would know every rule and condition that has to be met in the project, which gives him a sense of security. So even if both AI-based and conventional code analysis tools are both using the same principles in that they are made out of rules (created either through ML or static rules) and that both look through lines to check if it does something stupid and perform a "*static code analysis*" technically, they are only transparent for the developer in the conventional code analysis tools, he meant. He implied that AI-based tools without transparent rules and conditions did not give him any sense of security. Today he would not dare to run only Amazon CodeGuru or DeepCode as sole code analysis tools in project *T*, interviewee T1 added.

Both DeepCode and Amazon CodeGuru alerted him with the issue about "*I think this method or class is too big, you know*", which is easy to say but not easy to fix, he commented. He added that complexity is difficult to manage in software as well, you want it low, cannot measure it in a reasonable way and there are no general rules for it, interviewee T1 mentioned. At the same time, AI does not necessarily know why something is wrong, he added. He would not like to stretch the boundaries in that the AI would whitelist more and more things, which in the end might lead it to think that big classes are great. Still, there is certain code that is structured in a particular way that are complex, that there is nothing better to adjust it to (by current standards), he stated. Statistics classes is one example that interviewee T1 provided.

An interesting point that interviewee T1 suggested was that another good use case for these AI-based code analysis tool might be to fine-tune their actual conventional code analysis tools already in use, or that they possibly would improve each other. This is an interesting view point because it perhaps would give the best of two worlds. Perhaps a tool using both conventional and AI-based techniques together could bring the advantages of both sides to the surface without any apparent disadvantages. An advantage related to e.g. comprehension perhaps could be that the conventional techniques would explain why an issue was bad and what kind of rule triggered an issue, while an AI-based technique could display how many others had solved a similar problem.

**Robustness, Stability & Reliability**. Regarding the accuracy of both conventional and AI-based code analysis tools interviewee T1 stated that there always will be false positives. Some problems were just contextual, he added. Interviewee T1 pointed out that for conventional static code analysis tools someone needs to set it up with these rules and even if a lot of it is automated today, someone still needs to fine-tune it. In contrast, he meant that for AI-based tools it is more like "*You get what you see*". AI-based tools could also be used to fine-tune the static code analysis tools or vice versa, he pointed out. "*Perhaps some screws are tightened too tight or too lose*". For DeepCode and Amazon CodeGuru there is no way to *verify* the accuracy of the tool if there is no way to test it (also previously mentioned in *Comprehension*), according to interviewee T1. He further elaborated that because of they being very test-driven in project *T*, that there is no test to verify such changes against. In addition, previously in *Comprehension* interviewee T1 compared knowing how a tool worked through transparent rules with insurance, that gave him a feeling of security. The perceived reliability for such tool is likely higher than for AI-based tools without transparent rules. The combination of it being hard to verify (test) the accuracy of these tools and uncertainty about how these AI-based tools worked (no corresponding transparency or rules) likely negatively affected how accurate or reliable he perceived these tools to be. Related to what he said previously that some problems were just contextual and that there would always be false positives, interviewee T1 added that what both conventional and AI-based tools detect as critical vulnerabilities often might not be that dangerous, except for if it e.g. has to do with public API access or similar. If it regards something internal it would probably not be *that* critical, according to interviewee T1.

Interviewee T1 seemed content with the results the AI-based tools had provided. He commented that the quality for the AI-based tools he had tried in this case study was good and around the same as if he would compare to the conventional deterministic tools that they already, but added that it had to be further investigated. As previously mentioned in *Work Process Interplay* interviewee T1 had planted a few different bugs (code smells and vulnerabilities) and one multiple-step bug , that he knew the static tools caught, to see if DeepCode and Amazon CodeGuru would catch them, too. He remarked that the AI-based tools did catch all of the four bugs he had planted. Interviewee T1 pointed out that Amazon CodeGuru found 10 so-called *recommendations*, most of which had with (cyclomatic) complexity to do. The other recommendations were not applicable to this project and what he would class as false positives, but *these* redommendations the tool had no chance knowing were false positives, according to interviewee T1. If he could, he would whitelist these false positives that Amazon CodeGuru detected. An example of a false positive Amazon CodeGuru detected was the file or class *SynchronizationStatistics*, which provided a good overview and contained all statistic related methods and values (at the same place), interviewee T1 mentioned. He motivated that having all the statistics related methods and values at the same place made it easier to maintain and keep track on, while the cost of the larger size was minimal, according to interviewee T1. He also told us that DeepCode found about the same kind of things as Amazon CodeGuru did. Except for the intentionally planted bugs, DeepCode had detected two issues that actually were false positives. One false positive depended on that DeepCode thought the project rendered HTML, which it did not, but interviewee T1 said he could understand why DeepCode thought that. The second one was technically correct but not applicable in that specific case (he did not elaborate further), interviewee T1 stated. The third and last tool, Debricked, found no vulnerabilities, which was in line with interviewee T1's expectations, he said. He compared DeepCode and Amazon CodeGuru to a static code analysis tool made by OWASP (which one he did not specify), and pointed out that they in the end did the same thing. The latter however, was a lot more needy and not as user friendly as DeepCode and Amazon CodeGuru were, he remarked. He thought that these AI-based tools probably would fit perfectly in a CI/CD chain.

Interviewee T1 pointed out that Debricked had an extra feature which summarised and gave an overview all the licenses used in a project. It would help make sure someone does not do anything stupid like selling their source code, he added. Regarding this feature he compared Debricked to another tool called *FossID*, which could also manage licenses. However he thought that *FossID* were feature rich but with poor user friendliness where thought it was hard to pinpoint specific features because there was a whole range, which he described as them *"going the wrong path"*. In contrast, he pointed out that Debricked were going in the right direction; They focused on user friendliness and key features. As a final note, interviewee T1 told that he did not have any problems, issues, errors or delays when using the AI-based tools.

**User Satisfaction**. Interviewee T1 mentioned that he could rate Debricked separately and the other two (DeepCode and Amazon CodeGuru) together because they basically did the same thing to him. Interviewee gave Debricked a rating of 75 out of 100. He told us that he might be a little biased as he knew or recognised that some of the people there were from LTH, Faculty of Engineering in Lund. The reason for the rating he gave Debricked was that it was smooth to use, simple, and has a clear scope of what it does. It was easy to get going with Debricked and it has nice user interface, he added. Interviewee T1 said that he was even more happy with the two other tools (DeepCode and Amazon CodeGuru Reviewer) as they are more complex. He said that for such complex tools a lot need to go right in order to get something useful out, which he remarked that it did. The tools did find the bugs that he planted, he told us, indicating that DeepCode and Amazon CodeGuru Reviewer had performed well. With his reasoning interviewee T1 said that for these two tools he wanted to crank the rating up to 85. After giving the rating however, he told us that perhaps he should draw down the rating because of these two tools transferring the source code to third-party. It was unclear if he actually did draw down the rating, we therefore estimate the score somewhere in between these two values (75 and 85), i.e. a rating of 80. An example of a tool that did not transfer source code was FossID, according to interviewee T1, which transferred hashes instead, but this tool had another purpose. He reasoned that it would be better if he could install DeepCode or Amazon CodeGuru Reviewer locally to avoid any upload of source code.

Interviewee T1 pointed out that all of the three tools (Debricked, DeepCode and Amazon CodeGuru) had gone way longer in their development than what he previously thought. He could argue both for and against to keep using these tools, he pointed out. One deciding point whether to keep using these tools would be the cost, which currently is unclear, he pointed out. He told us that he could not with a good consciousness tell his supervisor to buy these tools for all projects, because as of now they do not have any proper use for them. He further explained that you need to be able to measure their usefulness somehow. Right now they already use other tools in project T that covered the most, i.e. code analysis, dependency licensing, vulnerabilities, etc, according to interviewee T1. He would start looking if they have use of integrating AI-based tools however. New angles are fun and surely useful long term to them, he motivated further.

Interviewee T1 stated that he would recommend these tools to his colleagues as they were easy and simple to use, and that you get a lot out of them with quick feedback if you cannot be bothered with the time-consuming setup of SonarCloud and its rules. The AI-based tools are definitely better than nothing, he added. Interviewee T1 thought that these tools likely would fit perfectly in a CI/CD chain with the reason of them being so smooth to use and integrate. The creators of these tools must have thought of that, he remarked, another example being the feature of analysis of each pull-request that he knew of. Interviewee T1 believed that AI-based tools will be used a lot more. However he also stated that he did not feel any need to be an early developer. It could perhaps complement their existing tools.

Interviewee T1 mentioned that he without hesitation believed that these AI-based tools could reduce his time spent reviewing code. He further explained that is because after a *git push* the tool could complain (on a pull-request basis) before the reviewer even starts testing or reviewing the code. It could give the developer a quick feedback before a reviewer needs to find faults that the tool already might have found, he implied. SonarCloud did not work that way, he mentioned, SonarCloud analyses the code all the time. He pointed out that SonarCloud could stop builds or merges to the base branch, but that these are different things than getting a comment from an AI-based tool directly in a pull-request, according to interviewee T1. In the end Interviewee T1 stated that he thought AI-based tools are exciting, he was curious what they would really mean in practise, and that he was glad to have taken part in the case study.

# Chapter 6
# Discussion

In this thesis we have mapped existing AI-based tools for software development (RQ1) and investigated how these affect the engineers' work environment (RQ2) through a case study. We identified what challenges AI-based tools bring (RQ3) through analysis of what work environment factors that affected the engineers the most negatively. The map showed that some AI-based tools today are mature enough that they could be used in professional software engineering. Our case study findings suggest that introducing AI-based tools in the engineers' work environment affects comprehension negatively. Also, in most cases, the perceived reliability and trustworthiness of a conventional tool is found to be superior to that of an AI-based tool. To be able to answer each research question comprehensively, we discuss each of them in a separate section.

## 6.1 Map of ML-Based Tools (RQ1)

We identified a few themes from the map in Chapter 4 that concerns the state of AI-based development tools today.

**Deep learning**. Deep learning seems to be the most prevalent ML technique used among the identified AI-based development tools. This suggests deep learning could be the most useful type for these kind of tools, even if such tools differ greatly between each other. Deep learning generally requires lots of data to train on. Most of these AI-based tools seem to have done just that, training on enormous amounts of data, mostly on (thousands of) open source repositories. Another reason could simply be due to the popularity of deep learning today.

**Heavy development & maturity**. There is a large amount of AI-based tools on the market today. A strong majority of the tools in the map were developed within the last five years, whereas all of them in the last ten years. This suggests that the area is under heavy development. Some AI-based tools are mature enough that they could be used within professional software engineering today or in the near future. None of the professional software developers that participated in the case study had any previous experience with the tools we presented to them, which could be because commercial AI-based tools today are relatively unknown. We deem that most of the commercial tools in the map of AI-based tools are surprisingly mature, even though they were unknown for the case study participants (and perhaps in large, the world of professional software development). The creators of the identified tools in the map ranged from being small teams or individuals to big companies. It seems that the tools developed by dedicated teams or big companies are well maintained and today can offer a professional user experience. The commercial tools seemed more well maintained and therefore more likely to be used professionally within the field of software engineering than the non-commercial alternatives.

**New concepts**. For some of the AI-based tools there might not exist corresponding conventional alternatives to compare with. As an example we have code generation from either input-output examples, text, or hand-drawn images. The overall concepts of either analysing code or generating code, are typical characteristics of both AI-based and conventional tools. Another example is that some recent AI-based tools originating from research projects attempted to automatically solve compiler errors. There seems to be no commercial alternatives yet for AI-based tools that use these relatively new concepts, which suggests the area of AI-based tools within SE is relatively new and in the development stage. This suggests that there are likely more commercial AI-based tools testing new concepts to be developed within the next few years.

However, overall many AI-based tools used concepts that had been well known for the last decade for conventional tools like code completion or static code analysis. The AI-based tools using these well-known concepts seemed on first impression mature enough to match up well with corresponding state-of-the-art conventional tools today, however this would have to be investigated further (possible future work). An indicator to this could be that all of the commercial AI-based code completion tools, just like conventional tools, are available as extensions to most popular editors and IDEs. A few of the AI-based *code review* tools introduced the ability of analysing code on a pull-request basis, suitable for code reviews. During the case study we did not encounter any conventional code review tools having this ability (we did not investigate this further). The possibility of a code analysis on a pull-request basis is a good way to provide instant feedback for a pull-request to the developer whom can fix the detected mistakes before an actual reviewer would have to review it.

## 6.2 AI Impact on The Engineer (RQ2)

### 6.2.1 AI Impact on Work Environment

Our case study results in Chapter 5 show that the use of AI-based tools affect the engineers' work environment and the outcome of their work. How the work environment is affected is discussed in this section, mainly through the factors defined in Table 2.1 on page 17.

**Training, Support & Documentation**. Most of the interviewees throughout all of the projects thought that the training was sufficient. The training consisted of a 15-minute introduction with a PowerPoint presentation explaining what this case study is about and what tool(s) they were going to use. After the introduction each participant would receive an e-mail with a couple of paragraphs for instructions for logging in or setting up the tool up correctly for all participants. For example, after setting up Debricked we described the initial setup or login information about how to add a user in Debricked and a reference setup of Debricked pipeline in their documentation. When they had troubles with the documentation there were two reoccurring themes: *lacking clarification* and *how to practically solve vulnerabilities*. For example, the most common issue seemed to be how to solve vulnerabilities for dependencies to dependencies in Debricked. These dependencies were already detected using AI and the *solving* part does not consist of any use of AI. Often this was combined with a lacking documentation regarding solving them.

For introducing AI-based tools in the general work-force, a challenge might be to have clear and comprehensive documentation for AI-based tools. A solution to this problem could be to include real use cases in either the documentation or as a tutorial about how the tool works practically. In addition it would help if terminology that could be perceived as unclear would be clarified in a comprehensive and clear way. The unclear terminology could be part because of AI being used in some of them. For example, in Debricked the vulnerability scores were often perceived as unclear, of which one of them was based on AI.

**Learning**. The learning curve to using these tools did not seem to be hugely affected by the use of AI in the products. For some participants the AI-based tools worked well and for some it worked less well. Most issues in learning seemed to have to do with navigating the interface and solving vulnerabilities, according to the interviewees, however this was likely not due to AI. Lacking documentation, especially for a tool that is hard to comprehend, could be the cause for a possibly steeper learning curve. For the case when the interviewees tried to understand the scores in Debricked, either they seemed to understand none of the scores or all of them. As one out of two of these vulnerability scores were AI-based, AI did therefore not seem to affect the learning curve significantly. Several participants thought the interface was unintuitive but it did not seem to affect the comprehension significantly after learning the tool. Many participants still gave a high user satisfaction even though they complained about the unintuitivety of the user interface.

**Work Process Interplay**. There was not enough projects using the approach *code review* in order to reach definite conclusions, but for the two projects that used this approach we do have some indicators. AI did not seem to affect the workflow noticeably. A new non-AI technique that we encountered however for AI-based tools was to analyse code on a pull-request basis. AI-based tools unlikely affect the workflow much. Potentially in the future if AI becomes excellent in performing code reviews they could perform an initial code review before an actual reviewer needs to spend time on it.

**System Interplay**. It varied greatly between the projects about if there already existed any integrations to the project or what other tools were used. This could be due to developers being more likely to integrate additional tools if they already have a working pipeline. In project *M* they had no pipeline or system integration at all while in project *T* they had several code analysis tool surveying the source code automatically. However none of the conventional tools used by the interviewees contained the ability to provide code analysis results directly in a pull-request. This feature is supported by both Debricked and DeepCode. One thing we would like to mention is that the setup process for the AI-based tools seemed to be simpler compared to a tool where you e.g. need to set up all the rules yourself (e.g. project T and SonarCloud). The setup process generally seemed to be easier for projects that did not have any already existing pipeline, e.g. as the default configuration for Debricked found in the documentation did not integrate well with existing pipelines (unless someone knew how to 'translate' it to fit in their configuration). For new tools like DeepCode, workspace setting changes had to be done to a BitBucket workspace in order allow it for some repositories. It was not possible to perform on a single repository, but all repositories in a workspace were affected. New tools seems to be more likely to have problems integrating if they are not 'official' apps, but 'new' does not necessarily mean that it has AI. In addition DeepCode wanted to automatically scan all repositories it could access before asking the user of which repositories it wanted the user to scan, which could be viewed as obtrusive. This could also affect the trust a user has in the tool, however it is not because of AI. For most projects however, the system integration was planned beforehand, partly mitigating such issues. For example, we did not integrate to BitBucket if it was not suitable through the pipeline. AI does not seem to affect the system interplay much.

**User Control**. For the most part there did not seem to be any perceived lack of user control in the AI-based tools for the interviewees regarding the use of AI. The few issues encountered however regarded functionality like not being able to cancel a pull-request generation in Debricked, customising the overview, or occasionally inconsistent vulnerability scan results outside of the user's control (pipeline in project *CB*). These issues did not seem to have anything to do with AI, except for possibly the last issue, if using AI could have led to this behaviour. In project *T* interviewee T1 mentioned that without lists of transparent rules there is less control over which 'rules' the AI uses. He pointed out that using an AI-based tool might require less effort to set up than a conventional tool with a time-consuming setup with e.g. rules to set up, but at the same time AI-based tools seems to provide less control than conventional tools, like SonarCloud. This is very probably to be a cause of AI. With the appearance of AI-based tools there may be an additional choice for developers: to choose whether they want an out-of-the-box AI-based solution with less effort and therefore less control, or conventional tools with more required effort but at the same time more control. The added freedom of choice between these two alternatives compared to only conventional tools may save costs for certain projects choosing the former (less effort), unclear in the long run however.

**Comprehension**. We believe that one of the main factors affecting programmers is understanding the AI-based functionality exhibited by the tools, i.e. *Comprehension*. Initially, none of the interviewees understood how to solve vulnerabilities affecting sub dependencies to direct project dependencies in Debricked. The weak comprehension of these may be due to the AI functionality, that if it would have been a conventional tool perhaps it would have been more straight forward to solve. The suggestions made by AI perhaps were not comprehensive for solving vulnerabilities affecting sub dependencies to direct project dependencies. One interviewee pointed out that he did not understand the AI-based diagrams in Debricked which described partly how the AI-based vulnerability score were calculated. No other participant outside of interviewee CB1 mentioned the diagram at all during the interview, possibly because it was unclear this diagram was related to the AI-based vulnerability score. How the vulnerabilities received certain scores (AI-based or not) was a very common issue among the interviewees, however. In addition, some participants were unsure of how much value to put into these scores. Sometimes the suggestions provided by the tools were also hard to understand. Something that was very split between the interviewees were how intuitive the interface of the AI-based tools were to navigate. For example around half of the case study participants thought that Debricked was intuitive while the other half thought that it was not. It is unclear if this is because of AI. For conventional tools that were generally understood in how they worked, there were configurable rules or detailed documentation. This is something that may be of a challenge for AI-based tools. These AI-based tools evaluated in the case study had weaknesses in their documentation, perhaps because the tools they were new. The perceived understanding of the tool seemed to very much affect how reliably the interviewees perceived the tool to be.

One interesting occurrence was that for the case when the tool identified issues that the interviewees actually identified as false positives, it was almost always the case that they thought they knew why the tool would think that. Perhaps it was because the AI used was trained to detect many issues and with that came the price of many false positives. It could also be due to the participants being very knowledgeable about the projects they are working on, or both. Perhaps the use of AI specifically were the cause of such false positives, as conventional alternatives better might check if something actually is referenced. It could also lead to developers being more open to using an AI-based tool as they could perceive themselves to be better at code reviewing than AI. An aspect that interviewee T1 also told us previously about were that false positives also exist for conventional code analysis tools like SonarCloud. However we do not know if false positives are more or less common in AI-based tools compared to conventional code analysis tools. In project *M*, *H* and *T* the interviewees actually did not seem to perceive the false positives in a negative way. It could give a sense of security to the developer knowing that the AI cannot differ between real issues and false positives, in that the developer is still at least as relevant and necessary. This sense of security could let a developer know that the AI-based tool is not better than him, and that the developer is irreplaceable. Perhaps a growing skill in the future with these AI-based tools compared to today would be to detect false positives. Even if AI-based code review tools would facilitate the developers' work and save time - the traditional code review may still be a necessity.

We noticed that the perceived user satisfaction of the tools seemed to be higher when we set up the tools for the interviewees instead of solely being assisting the setup as most. We likely saved these interviewees time setting the tool up which likely made the results to be generated perceived easier. However if the setup was performed by the developer themselves it would probably lead to better understanding of how the tool worked and was supposed to be used. The setup process for AI-based tools could possibly be made simpler. If a tool was understood well, it seemed to make the interviewee think of it as more reliable and robust. As mentioned previously, Interviewee CB1 thought that some of the solutions suggested by the tool were hard to interpret. For one of the vulnerabilities, he did not understand the suggested solution at all, or interpreted it as there being no solution. Interviewee CB1 was not alone regarding having a hard time to interpret some suggestions. This could be due to the AI-nature of the tools, in that the explanations of the suggestions may or may not be generated through conventional means.

One way that AI might be able to increase the comprehension for the programmer is when the tool provided examples of how a vulnerability was detected or fixed in another project or how it could be exploited. DeepCode always provided examples of other open source projects that had had the same issues or vulnerabilities - and how they fixed them. Debricked always linked to where the vulnerability was found, and if possible, how it could be exploited in code, too. A comprehension-related strength of using AI-based tools seems to be that they have the ability to show how other developers have solved a similar issue. This was the case with both Debricked and DeepCode, where Debricked could sometimes show how a vulnerability could be exploited in a code snippet, while DeepCode could for an issue e.g. show how a few other open-source projects had fixed the same issues with showing before-after diffs. This could help developers to understand an issue better and also to grasp the solution. It could show a glimpse of what skilled open-source developers tend to do when they encounter certain issues, which could be valuable for junior developers.

It is not improbable that when several vulnerabilities affected one package that there might be vulnerability duplicates. Due to the data-gathering nature of AI such duplicates could be more probable than for conventional tools, possible weakening comprehension (or at least learning). What the tools were based on were commonly hard to understand. Only one out of the three scores that Debricked used was based on AI however, it is therefore unclear it that is because of AI or unclear or lacking documentation. One thing that interviewee T1 mentioned was that for conventional tools like SonarCloud you were often able to filter for certain (transparent) rules or criteras. He previously mentioned that for AI it is more of a whole package that *"you get what you see"*. He commented that he did not find any way to filter like this in DeepCode or Amazon CodeGuru, and it would surprise him if AI-based tools would be able to filter like that, too, he mentioned. Not being able to filter part of the AI results might make comprehension more difficult. Understanding if a detected issue actually is an issue, may be problematic if using an out-of-the-box tool without any transparent rules. One things that could affect the comprehension were how well an AI-based tool could be able to explain *why* an issue was an issue.

**Robustness, Stability & Reliability**. We believe that the perceived reliability of a tool's exhibited AI-based functionality to be another major factor i.e. *Robustness, Stability & Reliability*. How reliable a tool is perceived to be is dependent on many factors, e.g. the accuracy, how consistently the tool behaved, negative user experiences and trustworthiness. Understanding how a tool works seemed to be related to its perceived reliability. Generally the users that seemed to have a better idea of how AI or ML worked during the interviews seemed more prone to find it reliable. However, another issue about using the AI-based tools are that it is very hard to measure how accurate they are (as previously mentioned by interviewee T1). When Debricked showed vulnerabilities for packages that the interviewees knew were outdated it seemed to agree with their expectations, and they probably found it more reliable (in line with the already-given world views and expectations), because old packages in general are more prone to being vulnerable. An AI-based tool detecting issues that a developer already knew about could have led to a sense of reliability.

One interesting indication that interviewee T1 hinted at was that the results that the AI-based code analysis tools did, seem to be comparable to the conventional code analysis tools already in use at that project. This would be very interesting for our problem statement had it been investigated further. This could be a possible future work.

**Ease of Remembering**. This was a very minor factor that we were only able to touch on the surface once in project *GA*, where they expected it to not be a problem outside of shortly having to think through in how to use it.

**User Satisfaction**. User satisfaction appears to be strongly affected by a user's comprehension and perceived reliability for an AI-based tool. How well a user thought of an AI-based tool and if they would like to keep using it seemed to be one of the main and most telling factor if an AI-based tool had a positive effect on a user's work environment. This factor could be viewed as a general summary of how the developer was affected by the other factors. *Comprehension*and *Robustness, Stability & Reliability* seemed to be the two main factors that affected user satisfaction the most, but most if not all other factors probably weighed in to a less significant degree. The users that were the most positive in comprehension and thought the AI-based tools were reliable, generally seemed to give a lot higher user satisfaction score. These two factors might be the two biggest challenges for AI-based tools. When the understandability was seemingly lower, the interviewees generally tended to give a lower user satisfaction rating of the tools. The average user rate for Debricked (if they would rate the tool between 0 and 100) was 77.5. The average user rating for DeepCode was estimated to 65. Both of these rating are quite high. The difference in these rates could be due to the different problems that these tools solve, it could also be due to differences in e.g. how comprehensive the tools were to use.

### 6.2.2 AI Work Impact

How the work is impacted in terms of code quality, code design, bug management and proportion of time a programmer could save is discussed in this section. To answer how the engineer's work and efficiency is affected by AI-based tools (RQ2) in an comprehensive way we will discuss four different sub themes: code quality, code design, bug management, and proportion of work time a programmer can save.

The results from the case study suggest that AI-based code review tools indicates to possibly be able to compare with conventional code review tools in that AI-based tools potentially can detect many issues that a conventional code review tool identifies. This is something that can be investigated further. Regarding the amount of code generation tools it is probable that in the future AI-based tools can *at least* provide a rough template or code skeleton of a quick prototype from an image, however the degree of quality of maintainability of the generated code were not investigated.

An interesting feature these AI-based tools brought, that is not any AI-based technology, is code review analysing code per pull-request, which was previously not encountered. Perhaps using AI or ML allowed for this feature more easily than already existing conventional tools.

#### Code Quality & Design

It is not entirely clear if using AI-based would lead to higher quality or better code design than compared to using conventional tools. It is clear however that the currently best AI-based tools could provide improvement suggestions that are at a similar level to what a conventional tool might suggest. An example was Debricked that there in these projects usually did not exist any conventional alternative to, where it actually did find a lot of (previously unknown to the developers) vulnerabilities. Amazon CodeGuru and DeepCode sometimes did suggest an issue regarding complexity of a program, which could be easy to detect but not easy to fix. For the cases when an AI-based tool did not find many issues it is unclear if the code already had a good quality or if the AI just was "bad". However, in no circumstance during the case study can we say that some AI in an AI-based tool actually behaved very poorly - which is a good indicator that these tools might be relatively mature.

#### Efficiency

At present it was clear that no AI-based tools saved or reduced any work time by any participants, instead the AI-based tools acted like an extra check for issues in the code. However, AI-based tools could save time long term through preventive measures, e.g. through fixing vulnerabilities found in Debricked. It is possible that conventional tools could work similarly well, however with AI comes the ability to manage huge amounts of data. As conventional tools are widely used today, there is no apparent disadvantage of at least complementing the current conventional tools with the AI-based ones. AI-based tools could lead to better initial efficiency than conventional tools, as it could be faster to getting started because of their tendency to be *"out-of-the-box"* solutions, while conventional tools often need to be configured.

### Bug Management

Debricked often found issues that almost all of the projects had no knowledge about. However, most projects had not encountered any conventional product tailored to dependency vulnerabilities previously. Amazon CodeGuru and DeepCode seemed to be able to find bugs comparing to current state-of-the-art conventional code analysis tool SonarCloud. In project *T* interviewee T1 had included several bugs that he knew the conventional static code analysis tools did find, and all of the issues were actually detected by DeepCode and Amazon CodeGuru. The conslusion seems to be that the current most popular AI-based code analysis tools seems to be at a similar level to conventional code analysis tools. It is uncertain however if AI-based tools alone long term would yield the same results as conventional tools. The advantage that AI-based tools provide for bug management could be that it potentially could show the reviewer where an issue was encountered followed by how other developers had fixed the same bug. This was unheard of for conventional tools, however they have the advantage of which transparent rules are broken.

## 6.3 Challenges (RQ3)

In this section we bring up what the major challenges for integrating AI-based tools are. Our third research question (RQ3) asked what challenges AI-based tools may bring. The challenges awaiting for introducing AI-based development tools likely depends mostly on trust (in a tool), understanding of how such tools works, and positive user experiences which could reinforce the user's trust in a tool. Not being able to understand exactly how an AI-based tool works (compared to e.g. transparent rules on a conventional tool) is a hinder for trusting that it does everything it should do - even if it supposedly can detect a large part of it. AI is no guarantee like a proper conventional tool that it will find *all* issues.

It is possible that for enterprise solutions where a tool could be trained on company repositories privately would require further education for the user, or at least time spent reading or learning about a given AI-based tool and its setup. Another thing that could be a hinder for using AI-based tools is where the product is made by a relatively new and untrusted company, compared to big, reputable companies such as Microsoft, Snyk and SonarSource. For a company to use an AI-based tool they would need to trust that the source code does not end up leaked or used for malicious purposes. Even though interviewee *T1* in project *T* were amazed by the results from the AI-based tools, he stated that he currently would not dare to use AI-based tools exclusively. The conventional tools he used provided him with a sense of security because he knew exactly how they worked and when rules would break. It is difficult to assert how or if AI-based tools could provide a similar sense of security.

Acceptance in using solely AI-based tools is seemingly a hard challenge for commercial projects. A more reasonable first step would be to use AI-based tools as a complement to conventional tools that may already exist in a project. It was also not always clear in all cases if the integrity of the developer was disturbed, e.g. if a tool would learn from your own code (or that it was already a fully-trained model being used), or if any other information is gathered by the company when using their tool.

# Chapter 7
# Conclusions

Artificial intelligence is undeniably going to affect the software engineering field. Within what aspects and to what degree is uncertain. The motive of using AI within software development would be higher quality code or increased efficiency by facilitating the developer's work tasks.

In software development there is a wide range of conventional developer tools that already assist developers in their tasks. A few examples of conventional tools are integrated developer environments (IDEs), static code analysis tools and testing tools. Two essential work tasks for most professional software development teams are code review where a reviewer reviews a change that another developer has made and programming. The purpose of performing code reviews are among others for maintainability, high quality code, and for knowledge-sharing. There is little previous research on what AI-based software development tools exist and how they affect the developer and their work. There have been previous work on how ML could be applied in SE, e.g. detection of faults or code smells. Digital tools also affect the work environment, and we wanted to find out what work environment factors affect us the most and how when using AI-based development tools would affect us. The research questions were to investigate what ML-based development tools that exist (RQ1) and how they affect the engineers and their work (RQ2). We also identified what challenges AI-based tools bring (RQ3). We investigated what ML-based tools there are (RQ1) by performing a literature review. This followed a case study with interviews that investigated how a few of these tools affected professional developers work environment (RQ2) at the case company. In order to conduct the case study we performed another literature review of how the digital work environment is affected by digital tools, of which we identified common factors to be studied of the case study participants. The participants took part in surveys and interviews, which we transcribed and then analysed. By analysing our results from RQ2 we were able to deduct what challenges are needed to be overcome in order to bring AI-based tools to professional software development (RQ3).

Our findings from the map (RQ1) suggest that the area is under heavy development as most tools are less than five years old, and all of the tools less than ten years old. However there is indicators that some tools may be mature enough to be used in some professional software development projects. This is because several (but far from all) of the case study participants were satisfied with the ML-based tools used in the case study, although only two case study participants in the same project kept using them in their professional software projects after the case study. Our case study findings (RQ2) suggest that development tools based on ML today are less understood and perceived to be less trustworthy than corresponding conventional tools. AI-based tools proved seemingly easier to set up than some conventional tools, however compared to conventional tools the user often do not know how AI-based tools work other than the results they provide. Lacking comprehension and trustworthiness of AI-based tools are challenges we deem crucial to overcome in order to successfully introduce such tools in professional software development (RQ3). One way to reduce the comprehension gap identified could be to improve the documentation of the tools. More integrations with online source code repositories could also avoid having to make compromises when setting up the tool. The different case study participants had varying experience within the SE field, we therefore deem the overall user satisfaction of each interviewee as a good indicator of how the AI-based tools can compare with the corresponding conventional ones in professional software development.

Our findings can help future research on how ML-based tools can improve and our map can act as a basis on what tools there are available today for future research. At present we suggest that the developers behind these AI-based tools focus on making the tools reliable, easy to learn and simple to use. If these parts would have been of no hinder for some participants, then we deem that the overall user satisfaction would have been increased significantly. Our work could lead to a greater spread of knowledge within the subject, it could give inspiration on what ML-based tools that do exist or are yet to be invented. This thesis acts as a future reconnaissance of how AI-based tools could change the software development profession in the future. The map could be further built upon or used as a basis for future research investigating ML-based development tools further. How the software engineer were affected and the challenged encountered may increase awareness of what issues to mitigate when introducing ML-based tools in professional software development. Further research could also be performed on how to overcome the identified challenges that using AI in software development tools may incur.

# References

[1] Hamza Abubakar, Mohammad S. Obaidat, Aaryan Gupta, Pronaya Bhattacharya, and Sudeep Tanwar. Interplay of machine learning and software engineering for quality estimations. In *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, pages 1–6, 2020.

[2] Inc. Amazon Web Services. Amazon codeguru | find your most expensive lines of code | aws. `https://aws.amazon.com/codeguru/`. Accessed: 2021-09-25.

[3] Inc. Amazon Web Services. Amazon codeguru faqs - amazon web services. `https://aws.amazon.com/codeguru/faqs/`. Accessed: 2021-09-25.

[4] Hany H Ammar, Walid Abdelmoez, and Mohamed Salah Hamdi. Software engineering using artificial intelligence techniques: Current state and open problems. In *Proceedings of the First Taibah University International Conference on Computing and Information Technology (ICCIT 2012), Al-Madinah Al-Munawwarah, Saudi Arabia*, volume 52, 2012.

[5] Atlassian. Bitbucket | the git solution for professional teams. `https://bitbucket.org`. Accessed: 2021-05-17.

[6] Noor Ayesha and N G Yethiraj. Review on code examination proficient system in software engineering by using machine learning approach. In *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, pages 324–327, 2018.

[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[8] Marco Barenkamp, Jonas Rebstadt, and Oliver Thomas. Applications of AI in classical software engineering. *AI Perspectives*, 2(1):1–15, 2020.

[9] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. `https://github.com/tonybeltramelli/pix2code`. Accessed: 2021-10-01.

[10] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2018.

[11] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[12] Laura Bordi, Jussi Okkonen, Jaana-Piia Mäkiniemi, and Kirsi Heikkilä-Tammi. Communication in the digital work environment: Implications for wellbeing at work. *Nordic Journal of Working Life Studies*, 8(Supp3):29 – 48, 2018.

[13] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, 43(1):56–75, 2017.

[14] Software Freedom Conservancy. Git. `https://git-scm.com`. Accessed: 2021-05-17.

[15] Software Freedom Conservancy. Selenium webdriver. `https://www.selenium.dev`. Accessed: 2021-05-17.

[16] Pribeanu Costin. A revised set of usability heuristics for the evaluation of interactive systems. *Informatică economică*, 21(3):31 – 38, 2017.

[17] Debricked. Your partner in open source | debricked. `https://debricked.com/`. Accessed: 2021-09-24.

[18] Debuild. Debuild - build web apps fast. `https://debuild.co/`. Accessed: 2021-09-24.

[19] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621. IEEE, 2018.

[20] Brian Dillon and Richard Thompson. Software development and tool usability. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, 2016.

[21] Facebook. Infer static analyzer | infer | infer. `https://fbinfer.com`. Accessed: 2021-05-17.

[22] Facebook. Paper artifacts for "aroma: Code recommendation via structural code search". `https://github.com/facebookresearch/aroma-paper-artifacts`. Accessed: 2021-09-23.

[23] Apache Software Foundation. Apache subversion. `https://subversion.apache.org`. Accessed: 2021-05-17.

[24] Eclipse Foundation. Enabling open innovation & collaboration | the eclipse foundation. `https://www.eclipse.org`. Accessed: 2021-05-17.

[25] Free Software Foundation. Cvs - concurrent versions system. `http://cvs.nongnu.org`. Accessed: 2021-05-17.

[26] OpenJS Foundation. Eslint - pluggable javascript linter. https://eslint.org. Accessed: 2021-05-17.

[27] The Apache Software Foundation. Maven – welcome to apache maven. https://maven.apache.org/. Accessed: 2021-05-17.

[28] César França, Fabio Q. B. da Silva, and Helen Sharp. Motivation and satisfaction of software engineers. *IEEE Transactions on Software Engineering*, 46(2):118–140, 2020.

[29] Fronty. Convert image to html css with ai: Fronty. https://fronty.com/. Accessed: 2021-09-24.

[30] GitHub. Atom. https://atom.io. Accessed: 2021-05-17.

[31] GitHub. Github. https://github.com. Accessed: 2021-05-17.

[32] GitLab. Gitlab: Iterate faster, innovate together. https://about.gitlab.com. Accessed: 2021-05-17.

[33] Gradle. Gradle build tool. https://gradle.org/. Accessed: 2021-05-17.

[34] Noah Gundotra. Code2pix: Generating graphical user interfaces from code. https://github.com/ngundotra/code2pix, Accessed: 2021-10-03.

[35] Ebba Håkansson and Elizabeth Bjarnason. Including human factors and ergonomics in requirements engineering for digital work environments. In *2020 IEEE First International Workshop on Requirements Engineering for Well-Being, Aging, and Health (REWBAH)*, pages 57–66. IEEE, 2020.

[36] Tracy Hall, Nathan Baddoo, Sarah Beecham, Hugh Robinson, and Helen Sharp. A systematic review of theory use in studies investigating the motivations of software engineers. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(3):1–29, 2009.

[37] Mark Harman. The role of artificial intelligence in software engineering. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 1–6. IEEE, 2012.

[38] Daniel Helgesson, Emelie Engström, Per Runeson, and Elizabeth Bjarnason. Cognitive load drivers in large scale software development. In *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 91–94. IEEE, 2019.

[39] Martin Höst, Björn Regnell, and Per Runeson. *Att genomföra examensarbete*. Studentlitteratur AB, 2006.

[40] Hussam Hourani, Ahmad Hammad, and Mohammad Lafi. The impact of artificial intelligence on software testing. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, pages 565–570. IEEE, 2019.

[41] Sublime HQ. Sublime text - a sophisticated text editor for code, markup and prose. https://www.sublimetext.com/. Accessed: 2021-05-17.

[42] JetBrains. Intellij idea: The capable & ergonomic java ide by jetbrains. https://www.jetbrains.com/idea. Accessed: 2021-05-17.

[43] Kite. Kite - free ai coding assistant and code auto-complete plugin. https://www.kite.com/. Accessed: 2021-09-23.

[44] Kite. What data does kite collect? - kite help desk. https://help.kite.com/article/84-kite-privacy-policy. Accessed: 2021-09-23.

[45] Elizabeth Knowles. *The Oxford dictionary of phrase and fable*. OUP Oxford, 2006.

[46] Steve Lamb and Kenny CS Kwok. A longitudinal investigation of work environment stressors on the performance and wellbeing of office workers. *Applied Ergonomics*, 52:104–111, 2016.

[47] Per Lenberg, Robert Feldt, and Lars G. Wallgren. Human factors related challenges in software engineering – an industrial perspective. In *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 43–49, 2015.

[48] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[49] Matt Mackall. Mercurial scm. https://www.mercurial-scm.org. Accessed: 2021-05-17.

[50] Microsoft. Azure devops services | microsoft azure. https://azure.microsoft.com/en-us/services/devops. Accessed: 2021-05-17.

[51] Microsoft. Prose - microsoft research. https://www.microsoft.com/en-us/research/group/prose/. Accessed: 2021-09-25.

[52] Microsoft. Transform sketches using ai into html. https://sketch2code.azurewebsites.net/. Accessed: 2021-09-23.

[53] Microsoft. Visual studio 2019 ide - programming software for windows. https://visualstudio.microsoft.com/vs. Accessed: 2021-05-17.

[54] Microsoft. Visual studio code - code editing. redefined. https://code.visualstudio.com/. Accessed: 2021-05-17.

[55] Microsoft. Visual studio intellicode - visual studio intellicode | microsoft docs. https://docs.microsoft.com/en-us/visualstudio/intellicode/. Accessed: 2021-09-24.

[56] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. Clever: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 153–164, 2018.

[57] OpenAI. Tabnine | gpt-3 demo. https://gpt3demo.com/apps/tabnine. Accessed: 2021-09-24.

[58] Bill Pugh and David Hovemeyer. Findbugs™ - find bugs in java programs. `http://findbugs.sourceforge.net`. Accessed: 2021-05-17.

[59] Margarete Sandelowski and Julie Barroso. *Handbook for synthesizing qualitative research*. springer publishing company, 2006.

[60] Emil Schutte. Code faster with ai code completions. `https://emilschutte.com/stackoverflow-autocomplete/`. Accessed: 2021-09-24.

[61] Semmle. Lgtm - continuous security analysis. `https://semmle.com/codeql`. Accessed: 2021-05-17.

[62] Amazon Web Services. Aws codecommit | managed source control service. `https://aws.amazon.com/codecommit`. Accessed: 2021-05-17.

[63] Yotam Shinan. The tabnine podcast - ai: Explained | tabnine blog, 2021. `https://www.tabnine.com/blog/eran-and-kyle-podcast-ep1/`. Accessed: 2021-09-24.

[64] Amanda Silver. Re-imagining developer productivity with ai-assisted tools - visual studio blog, 2019. `https://devblogs.microsoft.com/visualstudio/ai-assisted-developer-tools/`. Accessed: 2021-09-24.

[65] SonarSource. Automatic code review, testing, inspection & auditing | sonarcloud. `https://sonarcloud.io`. Accessed: 2021-05-17.

[66] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[67] Synopsys. Coverity scan - static analysis. `https://scan.coverity.com`. Accessed: 2021-05-17.

[68] Tabnine. Code faster with ai code completions. `https://www.tabnine.com/`. Accessed: 2021-09-24.

[69] Tabnine. Semantic completion. `https://www.tabnine.com/semantic`. Accessed: 2021-09-24.

[70] Tabnine. Gpt-3 version of tabnine is possible? · issue #281 · codota/tabnine · github, 2020. `https://github.com/codota/TabNine/issues/281`. Accessed: 2021-09-24.

[71] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 19–20, 2020.

[72] The JUnit Team. Junit 5. `https://junit.org`. Accessed: 2021-05-17.

[73] Uizard Technologies. Uizard. `https://uizard.io/`, Accessed: 2021-10-03.

[74] Uizard Technologies. Uizard research. `https://uizard.io/research/`, Accessed: 2021-10-03.

[75] VMware. Spring | home. `https://spring.io`. Accessed: 2021-05-17.

[76] Dror Weiss. Tabnine is now part of codota, 2020. `https://www.tabnine.com/blog/tabnine-part-of-codota/`. Accessed: 2021-09-24.

[77] Mark W Wiggins, Jaime Auton, Piers Bayl-Smith, and Ann Carrigan. Optimising the future of technology in organisations: A human factors perspective. *Australian Journal of Management (Sage Publications Ltd.)*, 45(3):449 – 467, 2020.

[78] Serdar Yegulalp. Python code completion gets an assist from machine learning | infoworld, 2019. `https://www.infoworld.com/article/3336484/python-code-completion-gets-an-assist-from-machine-learning.html`. Accessed: 2021-09-23.

[79] Du Zhang and Jeffrey JP Tsai. Machine learning and software engineering. In *14th IEEE International Conference on Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings.*, pages 22–29, 2002.

[80] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. *arXiv preprint arXiv:1809.04682*, 2018.

# Appendices

# Appendix A
# Interview Questions

**Introduction**.

- Tell me about yourself
  - Background
  - Current role or department of the case company
  - How long have you been working in that role?

**Code Review Specific**.

- Describe how you perform code reviews in your project.
  - Do you have guidelines for how code reviews are performed? Where do I find them?
  - What tools do you use when you perform code reviews?

**Training, Support & Documentation**.

- How was your experience when starting to use these tools?
- How was the introduction of Tool?
  - Sufficient? Needed? Helpful?
  - Wasn't there enough of anything?
- Was there something you needed to look up information or support with?
  - Where and how did you get that help? [documentation, co-workers etc]
    - Was it lacking, sufficient?
- Did you use the documentation?

  - How was it?
  - Enough documentation when needed, sufficient [or lack of info]?
  - Understandable at all? Useful?

**Learning**.

- Was it obvious from the beginning in how the tool is supposed to assist you?
  - Did it help you that way [that you expected]?
  - Why or why not?
- Did you have previous experience with Tool?
  - When did you hear or learn about it?
- How was your experience in learning how to use Tool?
  - Was it easy? Hard? Obvious?
  - Was it straight-forward in how to use the tool, since the beginning?
  - Were there any difficulties?
    * Which ones?
    * Do you have any suggestions for resolving these difficulties?
    * Feedback [to Company-of-Tool]?

**Work Process Interplay**.

- Describe your usage of Tool
  - When do you use it?
  - How often do you use it?
  - For how long periods of time?
  - What did the tool identify?
    * To what degree do you find the tool's code suggestions useful?
- Has using Tool made your work life easier or better?
  - Has it made you more productive?
- Is Tool intuitive to use?
  - Is the interface good?
  - Is there any cumbersome interaction with it?
- Does Tool fit well into your workflow
  - Is it adapted to your work tasks?
- Can Tool be better? In what?

- Is Tool always operating in a consistent way?
- Do you sometimes have to wait for the tool when working?
  - How long is this delay?
  - What do you do in the meantime?
- To what degree do you think the tool could perform a code review by itself?

**System Interplay**.

- How well do you think Tool integrated with your work environment i.e. other tools, pipelines and processes you have?
- Is there something Tool didn't work/integrate with well?
- Is there something that Tool worked extremely well with?

**User Control**.

- Does the tool have any settings or customization that you are aware of? Which ones?
- Do you perceive that you have control over the tool/work performed?
  - Provide an example please.
  - When and how it runs?
  - What part does it run on?
- Do you have freedom in deciding whether the suggestions provided by the tool are useful?
  - Can you describe how it works?
  - You can decide whether it is good or bad, to accept or deny it?
  - Are all suggestions completely optional?
  - Without any repercussion when denying a suggestion?

**Comprehension**.

- Do you understand how Tool works?
- Do you see why Tool arrived at the results that it gave you?
  - Can you describe how?

**Robustness, Stability & Reliability**.

- Do you find the tool reliable [with the suggestions it provides]?
  - Why is that?
- Is it accurate and consistent in its suggestions?

  - Can you give an example?

- Do you find it stable and robust?
  - Why?
- Did you encounter any problems or errors? When, how?
  - Did you understand why you got that error?
  - Did it take a lot of time to investigate these errors?

**Ease of Remembering**.

- Was it easy or hard to remember how to use the Tool in between code reviews?
  - How was your experience?
- How long [time] do you estimate there is between code reviews?
  - Or how many per day/week?

**User Satisfaction**.

- How satisfied are you with the tool between 0 and 100?
  - Why?
  - Why not 100?
  - Why did you pick that number?
- Are you going/do you plan to keep using Tool after this experiment?
  - Why or why not?
- Would you recommend Tool to your colleagues?
  - Why or why not?

**General Work Environment Questions**.

- Are you ever stressed during code reviews?
  - Has using this tool reduced it?

# Appendix B
# Survey Questions

Q1 How much time do you spend on one pull request in average (minutes)?
(Number answer)

Q2 What would you rate the overall code quality of [Project]?

- Very poor quality - Unstable and unreliable code, a lot of critical issues. Very hard to maintain. Basically the code is a mess.
- Poor quality - Not stable or reliable. There are a lot of issues, it is hard to maintain and a big refactor is needed, but it works.
- Fair quality - Mostly stable and reliable code with some known issues. Mostly maintainable
- Good quality - Stable and reliable code with only a few known minor issues. Somewhat easy to maintain
- Excellent quality - Stable and reliable code with no known issues. Well-tested and easy to maintain

Q3 To what degree do you expect AI-based tools to affect your code review productivity?

- Greatly complicate
- Complicate
- Neither complicates or simplifes
- Simplify
- Greatly simplify

Q4 To what degree do you expect AI-based tools to affect the quality of your review work?

- Greatly decrease quality

- Decrease quality
- Neutral, same quality
- Increase quality
- Greatly increase quality

Q5 How do you expect an AI-based code review tool to affect your work, e.g. regarding quality and productivity?

(Text answer)

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-10-28

**EXAMENSARBETE** AI-Assisted Programming
**STUDENT** Roland Veiderma Holmberg
**HANDLEDARE** Elizabeth Bjarnasson (LTH) and Björn Granvik (Softhouse)
**EXAMINATOR** Per Runeson (LTH)

# AI-Assisted Programming

POPULÄRVETENSKAPLIG SAMMANFATTNING **Roland Veiderma Holmberg**

AI-based technology is today being introduced to all parts of our society: everything ranging from vehicles and phones to finance and healthcare. Can AI help the software development of those, too? Could it make programming more efficient by predicting what programmers will type or to automatically find bugs? This may be the future or programming!

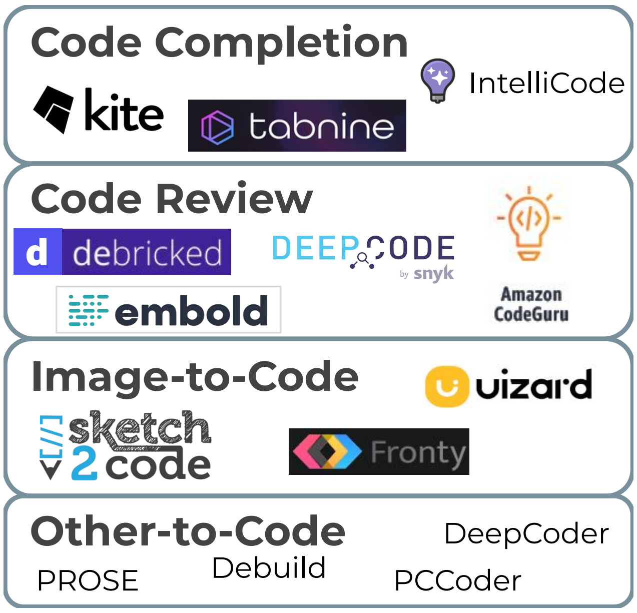
It is unclear how AI will impact software development, where the use of AI is quite new and undiscovered. There are some AI-based tools for programming, testing and finding bugs, but they are not used to any large extent. These tools are often new and under development.

In a study done together with Softhouse Consulting in Malmö, to investigated how AI will affect the work of software developers. We investigated the state-of-the-art w.r.t. research and currently available AI-based tools for software development. We produced a map of existing AI-based tools for programming and code review, and how these may assist developers. We performed a case study with IT consultants in the company to explore the implication of using AI based tools.

We observed that AI-based tools at present moment do *not* lead to any faster development, but that it slightly did increase software quality by identifying more potential vulnerabilities. Some AI-based code review tools were surprisingly quicker to set up compared to corresponding traditional tools, but at the cost of less transparency. Trust in AI-based tool showed to be one major challenge that needs to be overcome in order to successfully introduce it in the software industry.

How AI-based tools affect long term work environments (>1 year) is still unclear. AI-based tools are probable to require additional competence in the future.

In the map below you can find tools for code completion, code analysis or other intelligent prediction stuff - provided with pros and cons for each respective AI-based tool! Maybe there is one for you? If AI were to program itself in the future - this would be the middle step! A look into the future, if you will.


Code Completion
kite
tabnine
IntelliCode
Code Review
debricked
DEEPCODE by snyk
embold
Amazon CodeGuru
Image-to-Code
uizard
sketch 2 code
Fronty
Other-to-Code
DeepCoder
PROSE
Debuild
PCCoder