A Literature Survey of Software Analytics

Students and teachers of the 2018 IN4334 (Software Analytics) course @ TU Delft

2018-10-16

Contents

Chapter 1

Preamble

The book you see in front of you is the outcome of an eight week seminar run by the Software Engineering Research Group (SERG) at TU Delft. We have split up the novel area of Software Analytics into several sub topics. Every chapter addresses one such sub-topic of Software Analytics and is the outcome of a systematic literature review a laborious team of 3-4 students performed.

With this book, we hope to structure the new field of Software Analytics and show how it is related to many long existing research fields.

The IN4334 – Software Analytics class of 2018

1.1 License

Chapter 2

A contemporary view on Software Analytics

2.1 What is Software Analytics?

2.2 A list of Software Analytics Sub-Topics

Chapter 3

Testing Analytics

3.1 Motivation

Testing is an important aspect in software engineering, as it forms the first line of defence against the introduction of software faults Pinto et al. [**?**]. However, in practice it seems that not all developers test actively. In this chapter we will survey on the use of testing and the tools that make this possible. We will also look into the future development of tools that is done or required in order to improve testing practices in real-world applications. Testing is not the holy grail for completely removing all bugs from a program but it can decrease the chances for a user to encounter a bug. We believe that extra research is needed to ease the life of developers by making testing more efficient, easier to maintain and more effective. Therefore, we wanted to write a survey on the testing behavior, current practices and future developments of testing. In order to perform our survey, we formulated three Research Questions (RQs):

- **RQ1** How do developers currently test?
- **RQ2** What state of the art technologies are being used?
- **RQ3** What future developments can be expected?

In this chapter we will first elaborate on the research protocol that was used in order to find papers and extract information for the survey. Second, the actual findings for each of the research questions will be explained.

3.2 Research Protocol

For this chapter, Kitchenham's survey method [**?**] was applied. For this method, a protocol has to be specified. This protocol is defined for the research questions given above. Below the inclusion and exclusion criteria are given, which helped finding the rightful papers. After these criteria, the actual search for papers is described. The papers that were found are listed and after they are tested against the criteria that are given. The data that is extracted from these papers are list afterward. Some papers that were left out will be listed and the reasons for leaving them out will be given to make clear why some papers do not meet the required desire.

Each of the papers found was tested using our inclusion and exclusion criteria. These criteria were introduced to make sure the papers have the information required to answer the RQs while also being relevant with respect to their quality and age. Below a list of inclusion and exclusion criteria is given. In general, for all criteria, the exclusion criteria take precedence over inclusion criteria. The following inclusion and exclusion criteria were used:

- Papers published before 2008 are excluded from the research, unless a reference/citation is used for an unchanged concept.
- Papers referring to less than 15 other papers, excluding self-references, are excluded from the research.
- Selected papers should have an abstract, introduction and conclusion section.
- Papers stating the developers' testing behavior are included.
- Papers stating the developers' problems related to testing are included.
- Papers stating the technologies, related to testing analytics, which developers use are included.
- Papers writing about the expected advantage of current findings in testing analytics are included.
- Papers with recommendations for future development in the software testing field are included.

The papers used in this chapter were found by using a given initial seed of papers (query defined below as 'Initial Paper Seed'). From this initial seed of papers we used the keywords used by those papers to construct queries. Additionally, the references ('referenced by') and the citations ('cited in') of the papers were used to find papers. The query row of the tables describing the references, as found below, indicates how a paper was found. For queries the default search sites were Scopus,[1] Google Scholar[2] and Springer.[3]

The keywords used to construct queries in order to find papers were: software, test*, analytics, test-suite, evolution, software development, computer science, software engineering, risk-driven, survey software testing

The table below describes for each paper, which Query resulted in which paper being found. Each of the papers is categorized with a corresponding research question. In the table below, the categories per paper were added based on their general topic. These broad topics will be assigned to a corresponding research question. Categorizations are based on the bullet points extracted from each paper. These bullet points can be found in the appendix of this chapter in section *'Extracted paper information'*.

Category	Reference	Query	Relevant to
Co-evolution	Greiler et al. [74]	In 'cited by' of "Understanding myths and realities of test-suite evolution" on Scopus	RQ2, RQ3
Co-evolution	Hurdugaci and Zaidman [86]	Keywords: Maintain developer tests, 'cited by' in "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining" on IEEE	RQ2
Co-evolution	Marsavina et al. [120]	Google Scholar keywords: Maintain developer tests, in 'cited by' of "Aiding Software Developers to Maintain Developer Tests" on IEEE	RQ1
Co-evolution	Zaidman et al. [193]	Initial Paper Seed	RQ1
Production evolution	Eick et al. [64]	Referenced by: [109]	Discarded
Production evolution	Leung and Lui [109]	Initial Paper Seed	RQ3
Risk-driven testing	Atifi et al. [9]	In 'cited by' of "Risk-driven software testing and reliability"	RQ2, RQ3
Risk-driven testing	Hemmati and Sharifi [82]	In 'cited by' of "Test case analytics: Mining test case traces to improve risk-driven testing"	RQ3
Risk-driven testing	Noor and Hemmati [138]	Initial Paper Seed	RQ2, RQ3
Risk-driven testing	Schneidewind [164]	Scopus query: risk-driven testing	RQ3
Risk-driven testing	Vernotte et al. [186]	Scopus query: "risk-driven" AND testing	RQ2, RQ3

[1] https://www.scopus.com/
[2] https://scholar.google.com/
[3] https://www.springer.com

Category	Reference	Query	Relevant to
Test evolution	Bevan et al. [26]	Referenced by: [149]	Discarded
Test evolution	Mirzaaghaei et al. [132]	Google Scholar query: test-suite evolution	RQ2, RQ3
Test evolution	Pinto et al. [149]	Initial Paper Seed	RQ1
Test evolution	Pinto et al. [148]	Referenced by: [149]	RQ1
Test generation	Bowring and Hegler [34]	Springer: Reverse search on "Automatically generating maintainable regression unit tests for programs"	RQ2, RQ3
Test generation	Dulz [60]	Scopus query: "software development" AND Computer Science AND Software Engineering	RQ2
Test generation	Robinson et al. [159]	Referenced by [132]	RQ2
Test generation	Shamshiri et al. [166]	Google Scholar query: Automatically generating unit tests	RQ3
Testing practices	Beller et al. [25]	In 'cited by' of "Understanding myths and realities of test-suite evolution".	RQ1
Testing practices	Beller et al. [21]	Initial Paper Seed	RQ1
Testing practices	Garousi and Zhi [69]	Google Scholar query: Survey software testing	RQ1
Testing practices	Moiz [133]	Springer query: software testing	RQ3

3.3 Results

In this section the research questions will be answered. To answer these questions, information from the relevant papers are aggregated. The answers to each research questions are summarized in the conclusion.

3.3.1 (RQ1) How do developers currently test?

To answer RQ1, "How do developers currently test?", we first outline general test practices, then discuss the co-evolution of test and production code and finally, look into the use of Test Driven Development among developers.

3.3.1.1 How do we test?

For the quality of code, test coverage is a popular metric. Alternatives are, for example, acceptance tests, the number of defects in the last week, or defects per Line of Code (LOC) [69]. However, code coverage might not be the best indicator for the extensiveness of testing. For example, according to Beller et al. [22] a code coverage of 75% can possibly be reached with only spending less than a tenth of the total development time on testing. Another concern of using test coverage as a metric is the concept of treating the metric [33] , where developers try to uplift the value of code coverage by hitting many lines with only a few test cases. Marsavina et al. [120] observed that test cases were rarely updated when changes related to attributes or methods in the production code were made. Possible explanations for this are that these changes were not significant or the tests were too simple and were likely to pass. This also fits with the findings of Romano et al. [161], where they claim that "[d]evelopers write quick-and-dirty production code to pass the tests, do not update their tests often, and ignore refactoring."

Besides older tests rarely being updated for changed code, even new tests do not necessarily have the purpose of validating new production code lines. Pinto et al. [149] observed that a significant number of new tests

that are added, were not necessarily added to cover new code but rather to exercise the changed parts of the code after the program is modified. This finding fits with the observation of Marsavina et al. [120], who found that test cases are created or deleted in order to address the modified branches whenever numerous condition related changes are conducted in the production code base. Older production code lines, therefore, may stay untouched by any test cases. Lines uncovered by any traditional code coverage tool should be indicated and signaled to the developer. Therefore, developers should be aware of the fact that they did not cover some lines of their production code with any tests. It seems to be a deliberate action by most developers to not cover older production code lines. These lines might be 'too hard' to test, other lines may be easier to test, or developers do not seem to see the relevance of testing these uncovered lines of code. However, the most commonly used coverage metrics are branch coverage and conditional coverage [69]. As both branch coverage and conditional coverage require multiple different conditions for if-statements, it may possibly be that the absolute number of missed lines of production code by tests is very low but rather the number of missed conditions is higher.

3.3.1.2 Co-evolution

In a case study conducted by Zaidman et al. [193], there was no evidence found for an increased activity of testing before a release.

However, the study detected periods of increased test writing activity. These increased activities of writing test cases were found to be after longer periods of writing production code [193]. With a longer timespan of not writing tests, it can be concluded for these cases that the production code and test code do not gracefully co-evolve [193] [120].

3.3.1.3 Test-Driven Development (TDD)

We found different definitions for TDD across multiple studies. According to Zaidman et al. [193], evidence of TDD was found where test code was committed alongside production code, meaning that the methodology of TDD is used when production code was written before the respective test code. This is in contrast with the originally proposed constraint by Beck [19], where a line of production code should only be written after a failing automated test was written in advance. The confusion for the definition of TDD can also be traced back by the finding of Beller et al. [25], where programmers who claim they practice TDD neither follow it strictly nor practice it for all of their modification. A survey conveyed by Garousi and Zhi [69] on 196 respondents (amongst them managers and developers) indicated that with a ratio of 3:1 use Test-last development and Test-driven development respectively. This found ratio is in contrast with the numbers found by Beller et al. [25]; only 1.7% of the observed developers seemed to follow the strict TDD definition, where most of these developers only practice this strict definition in less than 20% of their time. However, it is important to mention that the survey done by Garousi and Zhi [69] only surveyed the subjects, which allows the confusion for the definition of TDD to play a major role in the results found.

3.3.2 (RQ2) What state of the art technologies are being used?

We will cover two research fields regarding testing analytics: test evolution and generation, and risk-driven testing.

3.3.2.1 Test Evolution and Generation

Pinto et al. [149] found the investigation of automated test repairing is not a promising research avenue, as these techniques would require manual guidance which could end up being similar to traditional refactoring tools. Nonetheless, more research is performed in this field since then. An approach for automatically repairing and generating test cases during software evolution is proposed by Mirzaaghaei et al. [132]. This

approach uses information available in existing test cases, defines a set of heuristics to repair test cases invalidated by changes in the software, and generate new test cases for evolved software. This properly repairs 90% of the compilation errors addressed and covers the same amount of instructions. The results show that the approach can effectively maintain evolving test suites and perform well compared to competing approaches.

While full automated test suite generation can not replace human testing entirely yet, Bowring and Hegler [34] introduced a tool that generates the templates for tests, which guarantees compilation, supports exception handling and finds a suitable location for the test. Developers still need to fix the test oracles themselves, but the template is there. The technique looks at the context in order to decide what template to use. Robinson et al. [159] created a regression unit tests generation tool. It is a suite of techniques for enhancing an existing unit test generation system. The authors performed experiments using an industrial system. The generated tests from these experiments achieved good coverage and mutation kill score, were readable by the product developers and required few edits as the system under test evolved. Dulz [60] found that by directly adjusting specific probability values in the usage profile of a Markov chain usage model, it is relatively easy to generate abstract test suites for different user classes and test purposes in an automated approach. By using proper tools, such as the TestUS Testplayer, even less experienced test engineers will be able to efficiently generate abstract test cases and to graphically assess quality characteristics of different test suites. Hurdugaci and Zaidman [86] introduces TestNForce (Visual Studio only), a tool to help developers identify unit tests that need to be altered and executed after code change.

3.3.2.2 Risk-driven Testing

The paper by Vernotte et al. [186] introduces and reports on an original tool-supported, risk-driven security testing process called Pattern-driven and Model-based Vulnerability Testing. This fully automated testing process, relying on risk-driven strategies and Model-Based Testing (MBT) techniques, aims to improve the capability of detection of various Web application vulnerabilities, in particular SQL injections, Cross-Site Scripting, and Cross-Site Request Forgery. An empirical evaluation shows that this novel process is appropriate for automatically generating and executing risk-driven vulnerability test cases and is promising to be deployed for large-scale Web applications.

A new risk measure is defined by Noor and Hemmati [138], which assigns a risk factor to a test case if it is similar to a failing test case from history. The new risk measure is by far more effective in identifying failing test cases compared to the traditional risk measure. Using this method for identifying test cases with a high risk factor, these test cases can for example be ran in the background while developing code, to find faults earlier. Furthermore, prioritizing these tests while running the entire test-suite could make the suite detect failing tests earlier and the developer can start fixing the faulty code right away.

3.3.3 (RQ3) What Future Developments Can Be Expected?

This section will elaborate on which future developments can be expected in the field of software analytics.

3.3.3.1 Co-Evolution and Test Generation

For understanding how test- and production code co-evolve and how tests can be generated to support developers, studies have been conducted [120, 149, 193]. Additionally a tool has been made in order to analyze and, consequently, better understand test-suite evolution [148]. For the time being the practical implications of this subtopic have mainly been sought in the repairing and generation of tests.

According to Pinto et al. [149] test repairs occur often enough to justify the development and research for automated repair techniques. M. Mirzaaghaei et al. [132] argue that evolving test cases is an expensive and time-consuming activity, for which automated approaches reduce the pressure on developers. Shamshiri et al. [166] argue that automated generation of unit tests does not end up generating realistic tests and that

the effectivity of developers writing manual tests is equal to developers using automatically generated tests. Therefore, they call for the use of more realistic tests. This suggests that automated test generation is still a topic of future interest, which will likely be researched in order to find a way to generate realistic tests.

3.3.3.2 Risk-driven Testing

Risk-driven testing is an area of recent attention. Researchers have been looking for methods that can either detect potential risks within the same project [138] [82] [186] or that can detect risks based on models carried over from one project to another [109] [9]. These techniques have been implementing history based prediction approaches.

In the future, we can expect more interest and research into risk-driven testing as allocating testing activities effectively will remain important due to testing efforts and developer time being expensive. This area will likely stay in its research phase for the next couple of years as effective measures for risk prediction are still being researched. This goes for measures within the same project and cross-project prediction. Given that the currently researched techniques regard history based implementations, it is likely that these techniques will be subject to further research later on.

3.3.3.3 Testing Practices

Research of several papers [69] [21] [25] has indicated that testing of any form is not as widely practiced as the status quo suggests. How the current state of the practice will change depends on various developments within the field. Tools will be created to assist the developer in writing quality code and tests, such as TestEvoHound as suggested by M. Greiler. [74]. As automated test generation becomes more effective this may reduce the need for developers to spend a lot of time on writing and maintaining tests. With the development of risk-driven testing, developers may also be able to focus on the parts that are likely to be the most important to address, which could lead to better time allocation. The status quo for how much time is to be expected to be spent on testing may also change, given automated test repair and generation techniques become effective and accessible.

3.4 Conclusion

In this chapter, three different research questions about software testing analytics were answered. (RQ1) How do developers currently test? (RQ2) What state of the art technologies are being used? (RQ3) What future developments can be expected?

Regarding the current testing practices of developers (RQ1), we found that developers do not seem to update their tests very often and when they do, it is because of a changed condition in production code lines. Furthermore, older uncovered production code lines are not likely to be covered in the end. Developers, thus, seem to ignore indications of their code coverage tools or do not seem to use any code coverage tool at all. Furthermore, developers do not seem to put a lot of effort into making sure the co-evolution of their production- and test code is done gracefully. They do, on the other hand, make sure their test code compiles when production code classes have been removed. However, testing is mostly done in longer periods of increased testing. The methodology of TDD also seems to be a confusing term for developers, as there is not enough clear guidance in the implementation of it. The actual ratio of TLD and TDD is, therefore, unknown but can be guessed with great certainty to be much lower for TDD than for TDD.

The current state of the art in testing analytics (RQ2) consists of research in co-evolution and generation of tests, and risk-driven testing. Approaches are proposed for automatically repairing and generating test cases during software evolution. While fully automated test suite generation is not there yet, a tool is introduced that generates the templates for tests, which guarantees compilation, supports exception handling and finds a suitable location for the test. In the field of risk-driven testing, new risk measures are defined which make

prioritizing certain high-risk tests able while running the entire test-suite, which could make the suite detect failing tests earlier.

For future developments (RQ3), further research can be expected on the front of automated test generation. Even with some discussion regarding the effectiveness of test generation, the field currently agrees that conducting research in order to find, especially, realistic ways of generating tests is worthwhile. We also found that risk-driven testing has been given more attention in the form of research recently. This subtopic is still in its research phase. It can be expected that research on the front of history based risk prediction methods will continue.

Chapter 4

Build analytics

4.1 Motivation

4.1.1 Introduction

Ideally, when building a project from source code to executable, the process should be fast and without any errors. Unfortunately, this is not always the case and automated builds results notify developers of compile errors, missing dependencies, broken functionality and many other problems. This chapter is aimed to give an overview of the effort made in build analytics field and Continuous Integration (CI) as an increasingly common development practice in many projects.

4.1.2 Continuous Integration and Version Control System

Continuous Integration in a term used in software engineering to describe a practice of merging all developer working copies to a shared mainline several times a day. CI is in general used together with Version Control System (VCS), an application for revision control that ensures the management of changes to documents, source code and other collections of information.

4.1.3 Build Definition

Build analytics covers research on data extracted from a build process inside a project. This contains among others, build logs from Continuous Integration such as Travis CI[1], Circle CI[2], Jenkins[3], AppVeyor[4] and TeamCity[5] or surveys among developers about their usage of Continuous Integration or build systems. This information is often paired with data from Version Control Systems such as Git.

4.1.4 Research Questions

We aimed to make a complete overview of build analytics field from analyzing current both state of the art and state of practice to inspecting the future research that could be done and finally conclude our survey

[1] See https://travis-ci.org/
[2] See https://circleci.com/
[3] See https://jenkins.io/
[4] See https://www.appveyor.com/
[5] See https://www.jetbrains.com/teamcity/

with the research questions that emerged after this exhaustive field research. To achieve a structural way of summarizing a field, we asked the following research questions:

RQ1: What is the current state of the art in the field of build analytics?

In section 4.3.1 we present the current topics that are being explored in the build analytics domain alongside with the research methods, tools and datasets acquired for the problems in hand and aggregate and reflect about the main research findings that the state-of-the-art papers display.

RQ2: What is the current state of practice in the field of build analytics?

Section 4.3.2 examines scientific papers to analyze the current trend of build analytics in the software development industry. We look at the popularity of CI in the industry and explore the increase in the use of Continuous Integration (CI) by discussing its ample benefits. Furthermore, it will discuss the practices used by engineers in the industry to ensure that their code is improving and not decaying.

RQ3: What future research can we expect in the field of build analytics?

In section 4.3.3 we will explore where new challenges lie in the field of build analytics. We will also show what open research items are described in the papers. This section ends with research questions based on the open research and challenges in current research.

4.2 Research Protocol

4.2.1 Search Strategy

Taking advantage of the initial seed consisting of Bird and Zimmermann[27], Beller et al. [23], Rausch et al. [157], Beller et al. TravisTorrent [24], Pinto et al. [146], Zhao et al. [196], Widder et al. [190] and Hilton et al. [83], we used references to find new papers to analyze. Moreover, we used academical search engine *Google Scholar* to perform a keyword-based search for other relevant build analytics domain papers. The keywords used were: build analytics, machine learning, build time, prediction, continuous integration, build failures, active learning, build errors, mining, software repositories, open-source software.

4.2.2 Selection Criteria

In order to provide a valid current overview of build analytics field, we selected only the relevant papers that were published after 2008, in other words we have not included papers older than 10 years. We had chosen 10 years as our threshold being inspired by "ICSE-Most Influential Paper 10 Years Later" Award. The only paper that does not obey to this rule is the cornerstone description of CI practices written by Martin Fowler, as we considered it important for us to see the practices evolution in build analytics field. Most of the papers we founded were linked to our research questions being reference in the sections bellow. From the selected papers, we omitted two papers, as they are case studies on a couple of project and do not introduce new techniques or applications.

See table 4.1 for an overview of the papers which were selected for this survey.

4.3 Answers

4.3.1 Build Analytics State of the Art

RQ1: What is the current state of the art in the field of build analytics?

The current state-of-the-art in the build analytics domain refers to the use of machine learning techniques to increase the productivity when using Continuous Integration (CI), to generate constraints on the configuration of the CI that could improve build success rate and to predict build failures even for newer projects with less training data available.

The papers identified using the research protocol defined section 4.2 that give us an overview of the current state of the art in build analytics field are:

- HireBuild: an automatic approach to history-driven repair of build scripts [79]
- A tale of CI build failures: An open source and a financial organization perspective [185]
- (No) Influence of Continuous Integration on the Commit Activity in GitHub Projects [11]
- Built to last or built too fast?: evaluating prediction models for build times [29]
- Statically Verifying Continuous Integration Configurations [163]
- ACONA: active online model adaptation for predicting continuous integration build failures [137]

The topics that are being explored are:

- the importance of the build process in a VCS project in reference [79]
- the impact factors of user satisfaction for using a CI tools in reference [190]
- methods from helping the developer to fix bugs in references [79], [184]
- predicting build time in reference [29]
- predicting build failures in references [163], [137]

The tools that are being proposed are:

- BART to help developers fix build errors by generating a summary of the failures with useful information, thus eliminating the need to browse error logs [184]
- HireBuild to automatically fix build failures based on previous changes [79]
- VeriCI capable of checking the errors in CI configurations files before the developer pushes a commit and without needing to wait for the build result [163]
- ACONA capable of predicting build failure in CI environment for newer projects with less data available [137]

4.3.1.1 Importance of the Build Process and CI Users Satisfaction

The build process is an important part of a project that uses VCS in the way that based on the findings of Hassan et al. [79], 22% of code commits include changes in build script files for either build working or build fixing purposes. Moreover, recent studies have focused on how satisfied the users of CI tools are, one paper by Widder et al. [190] analyzed what factors have an impact on abandonment of Travis CI. This paper finds that increased build complexity reduces the chance of abandonment, but larger projects abandon at a higher rate and that a project's language has significant but varying effect. A surprising result is that metrics of configuration attempts and knowledge dispersion in the project do not affect the rate of abandonment.

4.3.1.2 Patent for Predicting Build Errors

In reference [27], Bird et al. introduce a method for predicting software build errors. This US patent is owned by Microsoft. Having logistic regression as machine learning technique, the paper is able to compute the probability of a build to fail. Using this method build errors can be better anticipated, which decreases the time between working builds.

4.3.1.3 Predicting Build Time

Another important aspect is the impact of CI on the development process efficiency. One of the papers that addresses this matter is the one written by Bisong et al. [29]. This paper aims to find a balance between the frequency of integration and developer's productivity by proposing machine learning models that were

able to predict the build taking advantage of the 56 features presented in TravisTorrent build records. Their models performed quite well with an R-Squared of around 80%, meaning that they were able to capture the variation of build time over multiple projects. Their research could be useful on one hand for software developers and project managers for a better time management scheme and on the other hand, for other researchers that may improve their proposed models.

4.3.1.4 Predicting Build Failures

Moreover, the usage of automation build tools introduces a delay in the development cycle generated by the waiting time until the build finish successfully. One of the most recent analyzed papers by Santolucito et al. [163] presents a tool VeriCI capable of checking the errors in CI configurations files before the developer pushes a commit and without needing to wait for the build result. This paper focuses on prediction of build failure without using metadata like number of commits, code churn also in the learning process, but relying on the actual user programs and configuration scripts. This fact makes the identification of the error cause possible. VeriCI achieves 83% accuracy of predicting build failure on real data from GitHub projects and 30-48% of time the error justification provided by the tool matched the actual error cause. These results seem promising, but there is a need in focusing more on producing the error justification fact that could make the use of machine learning tools in real build analytics tools achievable and tolerated.

4.3.1.5 Prediction with Less Data Available

Even if there were considerable efforts in developing powerful and accurate machine learning models for predicting the outcome of builds, most of these techniques cannot be trained properly without large project past data. The problem that resulted from this is newer project being unable to take advantage of the research conducted before and having to wait until enough data from their project is generated to sufficiently train machine learning models from predicting the build outcome. In reference [137], the most recent paper of this survey which is only published as a poster in June 2018, Ni et al. address the problem of build failure prediction in CI environment for newer projects with less data available. It is using already trained models from other project with more data available and combined them by the means of active learning to find which of that models generalized better from the problem in hand and to update the model's weights accordingly. It is also aimed to cut the expense that CI introduce by reducing the label data necessarily for training. Even if the method seems promising, the results presented in the poster shows an F-Measure (harmonic average of recall and precision) of around 40% that could be better improved.

4.3.2 Build Analytics State of Practice

RQ2: What is the current state of practice in the field of build analytics?

Continuous Integration is a software engineering practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build which allows engineers to detect any bugs early.

An overview of Continuous Integration evolution from the introduction of the term to the current practices can be seen in the figure bellow:

The papers identified using the research protocol defined in section 4.2 that give us an overview of the current state of the art in build analytics domain are:

- Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects [83]
- An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software [157]
- Continuous Integration [66]
- Enabling Agile Testing Through Continuous Integration [173]

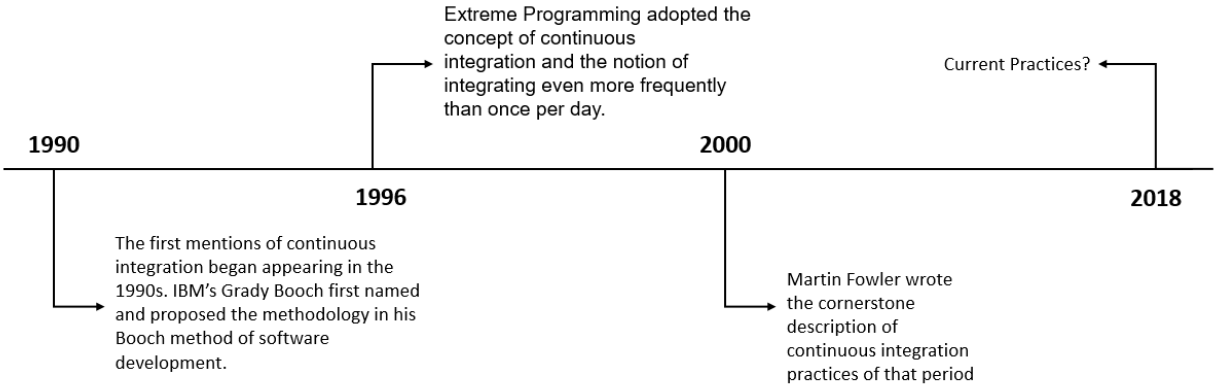

Figure 4.1: CI overview.

- Travistorrent: Synthesizing Travis CI and Github for Full-Stack Research on Continuous Integration [24]
- I'm Leaving You, Travis: A Continuous Integration Breakup Story [190]
- Continuous integration in a social-coding world: Empirical evidence from GITHUB [183]

The topics that are being explored are:

- Usage of CI in the industry by [83]
- Growing popularity of CI due to the introduction of VCS as suggested by [157]
- Common practices used in the industry exemplified by [66]
- Use of common CI practice in the agile approach presented by [173]
- Comparison between pull requests and direct commits to result in successful build as uncovered by [183]

4.3.2.1 Build Analytics Usage

A survey conducted in open-source projects by Hilton et al. [83] indicated that 40% of all projects used CI. It observed that a median project introduces CI a year into development. Furthermore, the paper claims that CI is widely used in practice nowadays. One of many factors contributing to this is explored by Rausch et al. [157]. The growing popularity of Version Control Systems (VCS) such as GitHub, and hosting build automation platforms such as Travis have enabled any business of size to adopt the CI framework. As suggested by Hilton et al. [83], the cost and time associated with introducing the CI framework is not enormous and the copious benefits far outweigh the resources required.

4.3.2.2 Build Analytics Practices

The CI concept, often attributed to Martin Fowler [66] , is recommended as best practice of agile software development methods such as extreme Programming [173]. He introduced many practices that are essential in maintaining the CI framework. Fowler and Foemmel [66] urges engineers to keep all artifacts required to build the project in a single repository. This ensures that the system does not require additional dependencies. In addition, he advises creating a build script that can compile the code, execute unit tests and automate integration. Once the code is built, all tests should run to confirm that it behaves as the developer would expect it to behave. In this way, we are finding and eradicating software bugs earlier and keeping builds fast. As explored by Widder et al.[190], one of the factors that lead to companies abandoning the CI framework is the complexity of the build. A good practice is to have more fast-executing tests than slow tests.

Furthermore, builds should be readily available to stakeholders and testers as this can reduce the amount of rework required when rebuilding a feature that does not meet requirements. In general, all companies should schedule a "nightly build" to update the project from the repository to ensure everyone is up to date. Continuous Integration is all about communication, so it is important to ensure that everyone can easily see the current state of the system. This is also another reason why CI works well in the agile industry [173]. Both techniques stress the importance of good communication.

The paper by Vasilescu et al. [183] studies a sample of large and active GitHub projects developed in Java, Python and Ruby. The paper finds that direct code modifications (commits) and more popular than indirect code modifications (pull request). Additionally, the notion of automated testing is not as widely practiced. Most samples in Vasilescu [183] study were configured to use Travis CI, however, less than half do. In terms of languages, Ruby projects are among the early adopters of Travis CI, while Java projects are late to adopt CI. The paper uncovers that the pull requests are much more likely to result in successful builds than direct commits.

4.3.3 Build Analytics Future Research

RQ3: What future research can we expect in the field of build analytics?

Currently research on build analytics is limited by some challenges, some are specific to build analytics and some are applicable to the entire field of software engineering.

The papers identified using the research protocol defined in section 4.2 that give us an overview of challenges and future research in the field of build analytics are:

- Built to last or built too fast?: evaluating prediction models for build times [29]
- Work Practices and Challenges in Continuous Integration: A Survey with Travis CI Users [146]
- Statically Verifying Continuous Integration Configurations [163]
- (No) Influence of Continuous Integration on the Commit Activity in GitHub Projects [11]
- The impact of continuous integration on other software development practices: a large-scale empirical study [196]
- Un-Break My Build: Assisting Developers with Build Repair Hints [184]
- Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub [23]

In Bisong et al.[29] the main limitation was the performance of the machine learning algorithm used. In the implementation R was used and it proved not capable of processing the amounts of data needed. This shows that it is important to choose the right tool when analyzing data.

In Pinto and Rebouças[146] it is noted that research is often done on open source software. There are still a lot of possibilities for researching on proprietary software projects.

Tools presented in papers might require a more large-scale and long-term study to verify that the tool presented keeps up when it is used[163].

Future research in build analytics branches in a couple of different topics. Pinto and Rebouças[146] proposes to focus on getting a better understanding of the users and why they might choose to abandon an automatic build platform.

Baltes et al.[11] suggest that in future research more perspectives when analyzing commit data should be considered, for instance partitioning commits by developer. It also notes the importance of more qualitative research.

Some open research questions from recent papers are the following:

- How do teams change their pull request review practices in response to the introduction of continuous integration? [196]
- How can we detect if fixing a build configuration requires changes in the remote environment? [184]
- Does breaking the build often translate to worse project quality and decreased productivity? [23]

- Could already trained models on projects with more data available be used to make accurate predictions on newer projects with less data available? [137]

From the synthesis of the works discussed in this section the following research questions emerged:

- What is the impact of the choice of Continuous Integration platform? Most of the research is done on users using Travis CI, there are many other platforms out there. Every platform has their own characteristics and this could impact the effectiveness for a specific kind of project.
- How does the platform or programming language influence effectiveness or adoption of continuous integration systems?
- How can machine learning methods be better applied in the field of build analytics in order to generate predictions that are easier to explain and thus that can be used in practice?

Appendix: Build Analytics

Table 4.1: Selected papers

Paper with reference	Source	RQ	Notes
1. Bird et al. 2017 [27]	Initial seed	RQ1	1
2. Beller et al. 2017 [23]	Initial seed	RQ3	-
3. Rausch et al. 2017 [157]	Initial seed	RQ2	-
4. Beller et al. TravisTorrent 2017 [24]	Initial seed	RQ2	-
5. Pinto et al. 2018 [146]	Initial seed	RQ3	-
6. Zhao et al. 2017 [196]	Initial seed	RQ3	2
7. Widder et al. 2018 [190]	Initial seed	RQ1	-
8. Hilton et al. 2016 [83]	Initial seed	RQ2	-
9. Vassallo et al. 2017 [185]	Ref 2	-	3
11. Hassan and Wang 2018 [79]	Ref 4	RQ1	-
12. Vassallo et al. 2018 [184]	Ref 2,3	RQ1, RQ3	-
13. Zampetti et al. 2017 [194]	Ref by 12	-	3
14. Baltes et al. 2018 [11]	GScholar Search	RQ1, RQ3	4
15. Bisong et al. 2017 [29]	GScholar Search	RQ1, RQ3	5
16. Santolucito et al. 2018 [163]	GScholar Search	RQ1	4
17. Ni and Li 2018 [137]	GScholar Search	RQ1	6
18. Fowler and Foemmel 2006 [66]	GScholar Search	RQ2	7
19. Stolberg 2009 [173]	GScholar Search	RQ2	7
20. Vasilescu et al. 2014 [183]	GScholar Search	RQ2	7

Notes

1. US patent owned by Microsoft.
2. Collaboration between universities in China, The Netherlands and The USA.
3. Not included in this survey as it did not introduce a new technique or practice.
4. Using search term “Github Continuous Integration”.
5. Using search term “Predicting build time”
6. Using search term “Predicting build failures”
7. Using search term “Current practices in Continuous Integration”

Chapter 5

Bug Prediction

5.1 Motivation

Minimizing the number of bugs in software is an effort central to software engineering — faulty code fails to fulfill the purpose it was written for, its impact ranges from slightly embarrassing to disastrous and dangerous, and last but not least — fixing it costs time and money. Resources in a software development life cycle are almost always limited and therefore should be allocated to where they are needed most — in order to avoid bugs, they should be focused on the most fault-prone areas of the project. Being able to predict where such areas might be would allow more development and testing efforts to be allocated on the right places.

However, reliably predicting which parts of source code are the most fault-prone is one of the holy grails of software engineering [63]. Thus it is not surprising that bug prediction continues to garner a widespread research interest in software analytics, now equipped with the ever-expanding toolbox of data-mining and machine learning techniques. In this survey we investigate the current efforts in bug prediction in the light of the advances in software analytics methods and focus our attention on answering the following research questions:

- **RQ1** What is the current state of the art in bug prediction? More specifically, we aim to answer the following:
  - What software or other metrics does bug prediction models rely on and how good are they?
  - What kind prediction models are predominantly used?
  - How are bug prediction models and results validated and evaluated?
- **RQ2** What is the current state of practice in bug prediction?
  - Are bug prediction techniques applied in practice and if so, how?
  - Are the current developments in the field able to provide actionable tools for developers?
- **RQ3** What are some of the open challenges and directions for future research?

5.2 Research protocol

We started by studying the initial 6 seed papers which were selected based on domain knowledge:

Reference	Topic	Summary
[75]	metrics validation	performance of object-oriented metrics
[37]	literature review	comparison of metrics, methods, datasets
[8]	literature review	comparison of models, metrics, performance measures

Reference	Topic	Summary
[62]	BP performance	benchmark for bug prediction, evaluation of approaches using the benchmark
[76]	literature review	influence of model context, methods, metrics on performance
[110]	case study	BP deployment in industry

We looked for additional papers with the following queries:

1. Keyword search using search engines (Google Scholar, Scopus, ACM Digital Library, IEEE Explorer). The search query was constructed so that the paper had to contain the phrase bug prediction, but also the other more general variants used in literature: *bug/defect/fault prediction*. The paper also had to contain at least one of following keywords: *metrics*, *models*, *validation*, *evaluation*, *developers*.
2. Filtering search results by publication date. We narrow the scope to *recent* papers, which we define as "published within the last 10 years" for the purposes of this review. Therefore we excluded papers published before 2008. However, there were a few exceptions for papers which had very high impact.
3. Filtering by the number of citations. We selected papers with 10 or more citations in order to focus on the ones that already have some visibility within the field. Also, there were some exceptions for every recently published papers.
4. Exploring other impactful publications by the same authors.

The papers chosen had to fulfill the above criteria.

Table 2. Papers found by investigating the authors of other papers.

Starting paper	Relationship	Result
[62]	is author of	[63]
[37]	is author of	[36] [38]
[155]	is author of	[154]
[72]	is author of	[71]

5.3 Answers

5.3.1 RQ1: Metrics

To find out what metrics are commonly used to build bug prediction models, we studied the papers that either focus their entirety or at least a section to the discussion of software metrics used for bug prediction. Based on the results and observations made in the relevant studies, we classify the software metrics most commonly used in two groups based on which aspect of a piece of software they measure:

- *The product*: the (static) code itself.
- *The process*: the different aspects that describe how the product developed through time. We include both simpler changelog based metrics that require different versions as well as metrics that require detailed process recording.

5.3.1.1 Product metrics

Different metrics incur different costs based on whether they require additional efforts in order to collect data and setup an instrumentation infrastructure. Measuring various aspects of what is already available —

a snapshot of source code — is an obvious starting point when building a set of predictive features and it turns out code metrics were the most commonly studied metrics in literature [37]. Traditionally, examples of metrics obtained by static analysis include *lines of code* (LOC) as well as *code complexity* (for example, McCabe and Halstead complexities), with the latter applicable for languages with a structured control flow and the concept of methods. The rationale behind using code complexity as a metric for bug prediction is that if code is complex, it is difficult to change and is therefore bug-prone [63].

Another group of more language specific metrics which became popular after 1990 are the *object-oriented* or *class* metrics which are based on the idea of classes and measure class-related concepts like *amount of methods*, *coupling*, *inheritance* and *cohesion* [75].

A drawback of static code metrics is that, the authors find that such metrics have high stasis which means they do not change a lot from release to release [154]. In turn, this can cause stagnation in the prediction models, which classify the same files as bug-prone over and over.

5.3.1.2 Process metrics

The other prominent group of metrics are the process metrics which are related to the software development process and the changes of the software through time. Additional infrastructure has to be in place in order to capture the process features and typically at least a version control system is required. One of the rationales behind using these metric in a predictive model is that bugs are introduced by changes in software and should be studied [63].

Some examples of process metrics include *code churn*, *number of revisions*, *active developer count*, *distinct developer count*, *changed code scattering*, *average lines of code added per revision* and *age of a file* [134]. There are also metrics based on *prior faults*, that argue files which in the past contained faults will probably have more faults in the future. It is shown that predictors based on previous bugs show better results than predictors based on code changes [155].

Other metrics are derived by looking at the source code change history as either *delta metrics* or *code churn* [152]. Delta metrics are not separate new metrics per se, but are derived from other metrics by comparing versions of software to each other.

Some studies found that process metrics outperform code metrics by comparing prediction results of both metrics [134]. It was also shown that process metrics are more stable across releases [154]. However, note that such results should be taken with a grain of salt; it was found that when comparing a selection of representative bug prediction approaches, the results considering metrics highly depend on the choice of learner and could not necessarily be generalized [63].

There are potential advantages of using more fine-grained source code changes, capturing them at statement level and including the changes' semantics [72]. This data can be obtained by building abstract syntax trees (ASTs) and using the changes required to transform one AST to the other as metrics. These models were found to outperform models based on other coarser metrics such as code churn; however, the gain in performance comes at the cost of extracting these fine-grained changes.

5.3.1.2.1 Developer-based metrics

Besides metrics from code repositories we can also extract some metrics from developers itself. It was shown that modules touched by more developers are more likely to contain more bugs [122]. Also, using developer based metrics in combination with other metrics can improve prediction results [122]. Furthermore, it turns out that developer experience has no clear correlation with how much faults they introduce [155].

Faults can also be predicted by tracking micro-interactions of developers [107]. The authors find that the most informative metrics are the *number of low-degree-of-interest-file editing events*, *editing and selecting bug-prone files consecutively*, and *time spent on editing*. They find that their prediction results exceed those of some product and process metrics.

Another set of metrics are scattering metrics, which uses data about how focussed a developers work is [54]. Structural scattering describes how structurally far apart in the project the code is. Semantic scattering measures how different the code being edited is in terms of implemented responsibilities, this is measured by textually comparing the code. The authors find they have relatively high accuracy and perform better than a baseline selection of code and process metrics.

5.3.2 RQ1: Models

In the section above we have seen different kinds of metrics, in this section we will show how these metrics are used to create models for bug prediction which can actually be used. We will also show some preliminary results of these models, however, as we will explain later on, it is not trivial to compare the results of different models.

5.3.2.0.1 History Complexity Metric [78]

Files that are modified during periods of high change complexity will contain more faults. Developers changing code during these periods will make mistakes, because they probably will not be aware of the current state of the code. This method is useful in practice because companies may not have a full bug history, while they probably do have history of code changes. Outperforms models based on prior faults, 15-38% decrease in prediction errors. However, D'Ambros et al. contradict this statement [62].

5.3.2.0.2 FixCache [99]

Maintains a fixed-size cache of files that are most likely to contain bugs. Files are added to cache if it meets one of the locality criteria, which are:

- Churn locality: if a file is recently modified it is likely to contain faults
- Temporal locality: if a file contains a fault, it is more likely to contain more faults
- Spatial locality: files that change alongside faulty files are more likely to contain faults

If the cache if full the least recently used file is removed from the cache. The size of the cache is set at 10% of all files. Hit rate of about 73-95% at file level, 46-72% at method level.

5.3.2.0.3 Rahman [155]

The Rahman algorithm is based on the FixCache algorithm, in their research they found that the temporal locality was by far the most influential factor. The Rahman algorithm is thus implemented based solely on that factor.

5.3.2.0.4 Time-weighted risk algorithm [110]

An algorithm based on the Rahman algorithm, it includes a weight factor based on how old a commit is, older bug fixing commits have less impact on the overall bug-proneness of the file. Instead of showing top 10% of bug-prone files, this shows the top 20 files.

5.3.2.0.5 Machine learning

With machine learning methods bugs can be predicted by software tools, these tools train on the code base, code history and other data from software repositories. Downside is that these methods give little insights in why a component is bug-prone [110]. Another downside is that this does not work for new projects since cross-project defect prediction does not yet give good results [197].

5.3.2.0.6 Analysis on Abstract Syntax Trees

Because ASTs give a more high-level description of the syntax of a program it could give more useful insights for bug prediction than other analysis methods. Downside is that ASTs is that dynamic programming languages cannot produce static ASTs. Wang et al. uses ASTs in combination with a machine learning approach for bug prediction. They also show that using ASTs can improve cross-project prediction results [188].

5.3.3 RQ1: Evaluations

In order to answer this questions we looked at the scientific papers that benchmark prediction models in any way, and specifically looked at the way these are evaluated. We found that most papers use a simple yet logical approach to benchmark an algorithm's performance. This approach is called Area Under the Curve (commonly referred as AUC), it consists of measuring the area under a curve which is known as Receiver Operating Characteristic (ROC). This process graphically draws the relationship between the accuracy, which is the ability of an algorithm to find all existing bug-prone files, and precision, which is how accurate the algorithm is at finding them (optimal would be no false negatives) [63].

Other authors use different methodologies in order to rank the algorithm's performance. Jiang et al. focuses solely on how to evaluate the algorithm's performance, and the researchers used the accuracy and precision metrics in numerical and graphical form, but they also bring to the table a graphic that helps identify where to spend the project resources in order to get a greater software quality [90]. Finally, another method is to use linear regression in order to qualify the fault prediction method's prediction power [62].

In conclusion, we can see that the most used methodology is AUC, this is because, citing Lessman et al.: "The AUC was recommended as the primary accuracy indicator for comparative studies in software defect prediction since it separates predictive performance from class and cost distributions, which are project-specific characteristics that may be unknown or subject to change." [108].

It is also worth noting the conclusion of Shepperd et al., in which they found that 30 percent of the variance of the algorithm's performance can be "explained" based on the research group, while the variability due to the choice of classifier is very small [167].

5.3.4 RQ2: Practical usage

There seems to be none to very little usage of bug prediction tools in the industry, even searching outside the scope of the scientific literature did not come up with anything. On GitHub we could only find a handful of repositories which implement a bug prediction tool, however, none of these are actively maintained any longer.

Furthermore, all the tools we did find were based on references [110] or [155]. Lewis et al. even mentions that their tool had no significant effect on developers [110]. Kim et al. says software managers could use a list of bug-prone files to allocate more quality assurance in some parts of the code, but again, no industry usage could be found [99].

We did find some reasons which could explain why there is no practical usage of bug prediction yet. Some issues we found with using bug prediction tools in practice have to do with the following properties that bug prediction tools should have in some cases:

- *Obvious reasoning*: If a tool has no clear reasoning users might quickly discard the tool, because they do not understand why a file is bug-prone [110].
- *Bias towards the new*: For some metrics this feature is required, because otherwise it would be impossible to 'fix' a file, it will remain to be bug-prone even if completely rewritten [110].
- *Actionable messages*: Messages shown by bug prediction tools must be actionable. The user must be able to perform an action to fix the problem with the file, otherwise users will not know what to do with the tool [110].

- *Noise problem*: It turns out that if data which is used for the historical changes contains noise this has a large impact on the performance of the model. Since the data is mostly mined from software repositories this will contain noise [98].
- *Granularity*: Most methods we found work on a class- or file-based granularity, having a lower level of granularity could improve the usefulness of the prediction for developers [71].

5.3.5 RQ2: Actionable tools for developers

The only paper that tackles this question is by Lewis et al., in which the authors deploy bug predicting software to developers workspaces and evaluate developers behaviour after and before having the software [110]. In this study they conclude currently, fault prediction does not provide actionable tools for developers.

To answer this questions we found no more information in scientific papers, so we had to look outside of the academic world in order to see if this techniques are being used by developers. After some research we concluded that bug prediction is not currently being used on developers, and we, as in reference [110], think that maybe bug prediction is supposed to be used for software quality in order to find the places where we should invest the project resources instead of helping developers write code without bugs.

5.3.6 RQ3: Open challenges and future work

We believe that the bug prediction field still has a lot of open challenges, for example, the use of developer-related factors in the predictions. Still no strong conclusions have been found, so this field requires a lot of further investigation to be able to find better models and useful applications of this techniques. All this challenges are anchored to a very hard to prove external validity.

In conclusion, bug prediction is still a partially explored area, and even though some algorithms have proven to be successful to some degree, we still lack more evidence and more reliable algorithms so they can be applied in the real world.

Chapter 6

Ecosystem Analytics

6.1 Motivation

In the modern day and age, it is popular to make use of external software or libraries in software projects to use the functionality (for example parsing JSON) of these libraries, without having to develop this functionality itself. For example, in 2016, the package manager npm recorded 18 billion package downloads in one month, according to [171]. Moreover, multiple languages, such as Python and Rust, provide package managers (pip[1] and Cargo[2] respectively) which can be used to easily manage this third-party functionality, as well as distribute it.

In parallel to this, the popularity of creating open source projects is on the rise as well. At the package repository npm only, 4,685 libraries were added in one week in 2016, according to [171]. On platforms such as GitHub[3], it is easy and quick to create a new software project, which can be developed, reviewed and used by the whole community. This development leads to more libraries being developed and being available for public use.

As a result of these two developments, software projects are increasingly becoming interdependent on each other. Inspecting the dependency relations between projects leads to a graph-like structure of software projects. In this structure,the nodes are the projects and the edges represent a dependency between two software projects. This structure is known as a *software ecosystem*. Messerschmitt and Szyperski in [131] state that a *software ecosystem* is "a collection of software products that have some given degree of symbiotic relationships." Another, similar definition is given by Lungu in [115]: "A software ecosystem is a collection of software projects which are developed and co-evolve in the same environment." Mens et al. in [130] extend this definition, "by explicitly considering the communities involved (e.g. user and developer communities) as being part of the software ecosystem." Stallman in [170] opposes the overall notion of calling this structure a software ecosystem: "It is inadvisable to describe the free software community, or any human community, as an ecosystem, because that word implies the absence of ethical judgment."

Although Stallman in [170] disagrees with the use of the term ecosystem, he does not necessarily disagree with the definition of the term. The definition which will be used in this chapter is the definition of Mens et al. in [130], since it captures the essence of the other two definitions, while adding the notion of the human communities alongside as well.

By performing analysis on these software ecosystems, the aim is to generate meaningful insights. These insights can then be used to improve the efficiency and effectivity of the software development process, as well as to learn to identify and inform about potential problems. For example, a warning could be displayed if a dependency has a security vulnerability.

[1] https://scholar.google.com/
[2] https://dl.acm.org/
[3] https://ieeexplore.ieee.org/

The field of research on software ecosystems, *ecosystem analytics*, focuses on performing such analysis. This chapter discovers what the current progress is in this field of research through a literature survey. This discovery is not limited to the theoretical perspective, but will uncover practical implications as well as the open challenges of the field. In order to describe each covered aspect, we have formulated three research questions:

- **RQ1**: What is the current state of the art in software analytics for ecosystem analytics?
- **RQ2**: What are the practical implications of the state of the art?
- **RQ3**: What are the open challenges in ecosystem analytics, for which future research is required?

Each of these research questions will be answered using recent papers written in this field of research.

This chapter is structured as follows. First, the research protocol is described in detail. This includes decisions on which papers are included in the review. After this, the research questions are answered using the previously stated set of papers.

6.2 Research Protocol

In order to review literature to answer the research questions given in the previous section, the survey method suggested by Kitchenham is used, conform to [101]. This method creates a systematic way to select a set of papers, which is relevant to the research question(s).

The search strategy, as described by Kitchenham [101], are usually iterative and benefit from consultations with experts in the field, among other things. Our search strategy can be split in three different types:

- the initial seed, given by an expert in the field, MSc. Joseph Hejderup
- a search using a digital search engine, namely Google Scholar[4]
- a selection of referenced papers within papers selected before in the above two searches

6.2.1 Initial seed

MSc. Joseph Hejderup has provided us with a total of thirteen papers, as shown in Table 4.1.

As each of these papers come from an expert in the field, each paper is assumed to be relevant to at least the field of software ecosystems. Because of this, each of these papers were judged on their relevance to either of the research questions. In Table 4.1, this relevance judgment is shown in the left column, since a paper is only selected, if the paper is indeed relevant. Table 4.2 describes the reason for which each particular paper is not selected for the literature survey.

6.2.2 Digital Search Engine

The second strategy type which is used to select relevant papers for this literature study, is by a digital search engine. In this literature survey, Google Scholar[5] is used. From the initial seed, common keywords were retrieved and the following queries have been used to search for relevant papers:

- “engineering software ecosystems” *(2014)*
- “software ecosystems” AND “empirical analysis” *(2018)*
- “software ecosystem” AND “empirical” *(2014)*
- “software ecosystem analytics” *(2014)*
- “software ecosystem” AND “analysis” *(2017)*

[4] https://ieeexplore.ieee.org/
[5] https://ieeexplore.ieee.org/

For each of these queries, the results were first filtered by the publish year. These are described by the italic year after each query above. The papers that are filtered are published earlier than the set publish year. These specific years were chosen since the survey focuses on the state of the art within the ecosystem analytics. The difference in years per query is a result from not finding relevant papers in more recent times (e.g. for "software ecosystem analytics", the year is *2014* because setting the filter to *2018* or *2017* did not result in relevant papers).

After this filtering, we first determined whether a paper was relevant to the literature survey by examining the title. If it was unclear whether the paper was indeed relevant by only looking at the title, the abstract of the paper was examined closely. On these two criteria, each of the selected papers were judged and ultimately selected. The selected paper using these method can be found in Table 4.3.

6.2.3 Referenced papers

The third method we use is by looking at the references found in the selected papers, using the two methods above. A selection has been made from these references. For these papers, the selection process is similar to that of the selected papers using the digital search engine; it is selected when both the title and the abstract are deemed relevant to the research questions. This has led to the papers in Table 4.4. being selected.

6.3 Answers

In this section, an aggregation of information, found in the papers, is presented. Each subsection of this section focuses on one of the three research questions posed in Section 1.

6.3.1 RQ1: What is the current state of the art in software analytics for ecosystem analytics?

To answer this research question, we examine the explored topics in ecosystem analytics. Moreover, we summarize which research methods, tools and datasets are being used to explore this topics.

6.3.1.1 Explored Topics

The main topics we explored in ecosystem analytics are the usage of trivial packages, (breaking) changes in dependencies and their impacts, quality of dependencies and dependency networks.

Trivial packages are a recurring topic in the area of software analytics, notably incidents like Left-pad, Schleuter [176], stress the possible impact trivial packages can have on a software ecosystem. Research in this topic explores the usage of trivial packages both quantitatively and qualitatively by analysing the usage of the trivial packages and the reasons why developers choose to use them respectively.

Breaking changes is a major topic researched by some papers. Much like trivial packages research is done both quantitatively and qualitatively in this topic. The impact of breaking changes and the way in which developers react to these changes are considered to be the notable problems in this topic.

Establishing a metric for the health of an ecosystem dependency is also a heavily explored topic. Researchers are trying to find ways in which a metric can be used to establish the quality of dependencies.

Dependency networks allow us to gain insight in the way in which ecosystems evolve over time.

6.3.1.2 Used Research Methods

The studied papers cover a plethora of research methods. These methods can be divided into two categories: quantitative and qualitative.

Many quantitative research papers analyse the data in a statistical manner, using software ecosystems as their data. The types of data depend heavily on the ecosystem used for analysis. Some papers go as far as using the source code of packages [3]. While other research focusses on the meta-data of software ecosystems, such as dependency networks [96]. Another recurring research method is survival analysis, as used by [53], which can be used to estimate the survival rate of a population over time. In software engineering this has been successfully applied to open source projects.

Some qualitative research has been done in software ecosystems to gain a better understanding in to the behaviour of the interactions between developers and software ecosystems. Some papers solely rely on the results of qualitative research whereas some papers use both quantitative research and qualitative research to triangulate their findings.

6.3.1.3 Used Ecosystems

As has been said above, the most common explored topic within the body of studied papers are topics relating to dependencies between different software projects. Therefore, it does make a lot of sense that the most common ecosystems used are those of package managers, since these package managers are central repositories from which the packages, as well as their dependencies, can easily be retrieved.

The most common used ecosystem in the studied papers is *npm*[6], as used by Abdalkareem et al. [3], Bogart et al. [31], Decan, Mens, and Claes [52], Kikas et al. [96], and Decan, Mens and Grosjean [53]. There are a few reasons why this is the most common used ecosystem throughout the papers. Firstly, npm is the largest software registry, containing more than double of the next most populated package registry in 2016 [171]. Moreover, npm is the package manager of JavaScript, which is the most used programming language according to a RedMonk survey [177]. Kikas et al. [96] explain that it is beneficial to use this ecosystem, because the majority of their packages are hosted on GitHub and Developers specify required packages in their project's dependency files. There are more ecosystems which have the same properties, such as RubyGems (Ruby) and Crates.io (Rust) [96].

However, these are not the only used ecosystems. Decan, Mens, and Grosjean [53] use the *libraries.io* dataset for their research, which includes seven different packaging ecosystems: Cargo (rust), CPAN (Perl), CRAN (R), npm (JavaScript), NuGet (.NET), Packagist (PHP) and RubyGems (Ruby). Decan, Mens, and Grosjean [53] therefore study the most different ecosystems in one paper, relative to the papers studied in this survey.

Apart from these datasets based on packages in package repositories, there are papers which use other datasets and therefore ecosystems as well. Bavota et al. [16] research the Apache Community and therefore uses a dataset corresponding to the full Java subset of the Apache ecosystem. Cox et al. [48] instead use the dataset of the Apache Maven Project to validate his created metric. Claes et al. [43] research package incompatibilities and therefore uses a ten year period of the Debian i386 testing and stable distributions. Robbes, Lungu, and Röthlisberger [158] opted for the Squeak/Pharo ecosystem, as they stated that this ecosystem would provide support for answering their research questions.

6.3.1.4 Main research findings

Based on the findings of Abdalkareem et al. [3] trivial packages make up 16.8% of the NPM packages. Even though 10% of NPM uses trivial packages only 45% of these trivial packages have tests.

Robbes et al. [158] mentions the large impact API changes can have on an ecosystem. Bogart et al. [31] studied the attitude of developers towards breaking changes in dependencies. Their main findings were that

[6] https://www.macrumors.com/2016/10/10/apple-dash-developer-fraudulent-reviews/

an ecosystem plays an essential role in the way we can deal with breaking changes. Both papers conclude that developers generally do not respond in time to breaking changes and as a result breaking changes can have a large impact on a software ecosystem. This conclusion is reinforced by the findings of Decan et al. [53], where frequent changes can lead to an unstable dependency network due to transitive dependencies.

Not only do developers not react in a timely fashion to breaking changes, Robbes et al. [158] also found that developers are also not quick to respond to API deprecation. Bavota et al. [16] suggests that updates should only be done when they consist of bug fixes, not API changes, to combat this issue.

Attempts have also been made to find a metric that establishes the 'health' of a dependency. Cox et al. [48] contributes to this by providing a metric to establish the freshness of a dependency.

An interesting finding in the topic of package dependency networks by Kikas et al. [96] is that ecosystems, over time, become less dependent on a single popular package.

6.3.2 RQ2: What are the practical implications of the state of the art?

In this research question we aim to find out the practical implications of the state of the art as discussed in the previous section. The papers discussed in this section are mostly case studies and we will summarize their findings.

From most papers we find that developers are slow when updating their dependencies, or sometimes they do not even do it at all. Hora et al. [84] suggest that a main reason for this is that breaking changes cannot be solved in a uniform manner throughout the ecosystem, but rather need a specific implementation for each system. We have also found that breaking changes are constantly introduced when dependencies are updated. According to Raemaekers, van Deursen and Visser [153], about 33% of releases, either minor or major, contain a breaking change that needs looking into. Breaking changes could pose compiling errors, thereby breaking the system that depends on it.

Developers tend to react poorly to changes in their dependencies; Kula et al. [104] have found that, of the 4600 surveyed projects, 81.5% of the projects contain outdated dependencies which can lead to security risks. Not only do developers not update their dependencies, according to an empirically study conducted by McDonnell, Ray and Kim [124] on the android API, they also do not update their codebase with respect to the changes introduced by dependencies.

Regarding ecosystem health, Constantinou and Mens [45] have researched which factors indicate that a developer is likely to abandon an ecosystem. Their study, which analysed GitHub[7] issues and commits, has found that developers are more likely to abandon a system when they 1) do not communicate with their fellow developers, 2) do not participate often in social or technical activities and 3) for an extended period of time do not react or commit any more. Another interesting characteristic about ecosystem health, studied by Kula et al. [103], is the way in which projects age over time. Their study found that the usage over time of 81.7% of 4,659 popular GitHub projects can be fitted on a function with an order higher than two.

Malloy and Power [117] have studied the transition from Python 2 to Python 3. Python 3 has been out since 2008, and the final Python 2 release was in 2010. Both are (almost) 10 years ago. Even though, during their study they have found that most Python developers choose to maintain compatibility with both Python 2 and Python 3 by only using a subset of the Python 3 language. Malloy and Power [117] state that they are severely limiting themselves in their language capabilities, by not using the newly developed features.

Another interesting topic of research is the impact tools can have on ecosystems. Among these tools are badges. Badges are annotations on software projects which display some information about a software project. One of these badges can warn developers about outdated packages. Based on the results of Trockman [182], badges can have a positive impact on the speed at which developers update their dependencies.

Overall we can conclude that there are a lot of improvements to be made. The current method that most users use to manage their dependencies is lacking. Whether it be updating late or not updating at all, there

[7] https://ieeexplore.ieee.org/

are many risks bound to this. Dietrich, Jezek, and Brada [58] have also found that there are a lot of problems in the Java ecosystem, and has posed a set of relatively minor changes to both development tools and the Java language itself that could be very effective. These improvements are highlighted by answering the last research question.

6.3.3 RQ3: What are the open challenges in ecosystem analytics, for which future research is required?

This research question gives insight in the current open challenges in the field of ecosystem analytics. It focuses on the challenges described in the studied papers.

The most common open challenge across almost all papers is the generalization of results. Most of the studied papers only use a single ecosystem on which they base their results. This in turn means that it is unsure whether these results hold for other ecosystems as well in a similar fashion. For example, Claes et al. [43] state that a possibility for future work is to investigate to what extent the findings can be reused in the framework of other package-based software distributions.

However, even if multiple ecosystems are researched within one paper, it shows to still not be enough to provide a generalization for each ecosystem. Decan, Mens, and Grosjean [53] state, after researching dependency network evolution for seven ecosystems, that they do not make any claims that their results can be generalized beyond the main package managers for specific languages. This is because Decan, Mens, and Grosjean [53] do not expect similar results for networks such as WordPress, as these packages tend to be more high-level (e.g. used by end users instead of reused by other packages). This is shown as well in the different results obtained by Bogart et al. [31], which shows that values differ per ecosystem. This overall shows that there is a lot of space for future research to be done in generalizing research beyond the already researched ecosystems.

Another persistent open challenge is the ability to determine the health of an ecosystem. Although Jansen [88] has provided OSEHO, "a framework that is used to establish the health of an open source ecosystem", Jansen [88] notes that "there is surprisingly little literature available about open source ecosystem health". Kikas et al. [96] agree, stating that a general goal is to provide analytics to maintainers about the overall ecosystem trends.

This challenge is related to determining a systems health. Kikas et al. [96] state that "a measure quantifying dependency health in an ecosystem should be developed". Moreover, according to Jansen [88], determining the health of a system from an ecosystem perspective is required to determine which systems to use.

Because of this, this challenge ties into the assistance while selecting packages as well. This challenge is about selecting the best dependency, according to the functionality needs of the existing application. Abdalkareem et al. [3] state that helping developers find the best packages suiting their needs need to be addressed. Kikas et al. [96] agree that another general goal is to provide maintainers with tools to manage their dependencies.

However, whenever these dependencies are chosen, another open challenge is to assist maintainers to keep these dependencies up to date. In order to find out when dependencies have to be updated, metrics have to be defined. Bavota et al. [16] state that their observations could be a starting point to build recommenders for supporting developers in complex dependency upgrade activity. Cox et al. [48] provide "a metric to aid stakeholders in deciding on whether the dependencies of a system should be updated". However, Cox et al. [48] state multiple refinements on this metric which could still be researched.

6.4 Appendix

Selected	Author(s)	Title	Year	Keywords
-	Abate, Di Cosmo, Boender, Zacchiroli [2]	Strong dependencies between software components	2009	
-	Abate, Di Cosmo [1]	Predicting upgrade failures using dependency analysis	2011	
+	Abdalkareem, Nourry, Wehaibi, Mujahid, Shihab [3]	Why do developers use trivial packages? An empirical case study on NPM	2017	JavaScript; Node.js; Code Reuse; Empirical Studies
+	Bogart, Kästner, Herbsleb, Thung [31]	How to break an api: Cost negotiation and community values in three software ecosystem	2016	Software ecosystems; Dependency management; semantic versioning; Collaboration; Qualitative research
+	Claes, Mens, DI Cosmo, Vouillon [43]	A historical analysis of Debian package incompatibilities	2015	debian, conflict, empirical, analysis, software, evolution, distribution, package, dependency, maintenance
+	Constantinou, Mens [45]	An empirical comparison of developer retention in the RubyGems and NPM software ecosystems	2017	Software ecosystem, Socio-technical interaction, Software evolution, Empirical analysis, Survival analysis
+	Hejderup, van Deursen, Gousios [81]	Software Ecosystem Call Graph for Dependency Management	2018	
+	Kikas, Gousios, Dumas, Pfahl [96]	Structure and evolution of package dependency networks	2017	
+	Kula, German, Ouni, Ishio, Inoue [104]	Do developers update their library dependencies?	2017	Software reuse, Software maintenance, Security vulnerabilities
-	Mens, Claes, Grosjean, Serebrenik [130]	Studying Evolving Software Ecosystems based on Ecological Models	2013	Coral Reef, Natural Ecosystem, Open Source Software, Ecological Model, Software Project

Selected	Author(s)	Title	Year	Keywords
+	Raemaekers, van Deursen, Visser [153]	Semantic versioning and impact of breaking changes in the Maven repository	2017	Semantic versioning, Breaking changes, Software libraries
+	Robbes, Lungu, Röthlisberger [158]	How do developers react to API deprecation? The case of a smalltalk ecosystem	2012	Ecosystems, Mining Software Repositories, Empirical Studies
+	Trockman [182]	Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem	2018	

Table 4.1: Papers provided by MSc. Joseph Hejderup. The first column describes whether the paper of the row will be used. A '+' means it will be used, a '-' means it will not.

Paper Reference	Reason not selected

Paper Reference	Reason not selected
[2]	This paper seems to delve more into one software project itself whereas we are more interested in the relationship between different software projects

Paper Reference	Reason not selected
[1]	Similarly to [2], we are more interested in the relationship between different software projects

Paper Reference	Reason not selected
[130]	We were in doubt over this one, it could be useful but we weren't convinced that it was. Since we already had a lot of material we decided to not use this

Table 4.2: Papers from the initial seed that were not selected for the literature survey, along with a specification of the reason why this is the case.

Author(s)	Title	Year	Keywords	Query Used
Decan, Mens, Grosjean [53]	An empirical comparison of dependency network evolution in seven software packaging ecosystems	2018	Software repository mining, Software ecosystem, Package manager, Dependency network, Software evolution	"software ecosystems" AND "empirical analysis"
[59]	Software engineering beyond the project – Sustaining software ecosystems	2014		engineering software ecosystems

Author(s)	Title	Year	Keywords	Query Used
Hora, Robbes, Valente, Anquetil, Etien, Ducasse [84]	How do developers react to API evolution? A large-scale empirical study	2016	API evolution, API deprecation, Software ecosystem, Empirical study	"software ecosystem" AND "empirical"
Izquierdo-Gonzalez-Barahona, Kurth, Robles [87]	Software Development Analytics for Xen: Why and How	2018	Companies, Ecosystems, Software, Measurement, Object recognition, Monitoring, Virtualization	software ecosystem analytics
Jansen [88]	Measuring the Health of Open Source Software Ecosystems: Beyond the Scope of Project Health	2014		"open source software ecosystems"
Kula, German, Ishio, Inoue [103]	An exploratory study on library aging by monitoring client usage in a software ecosystem	2017		"software ecosystem" AND "analysis"
Malloy, Power [117]	An empirical analysis of the transition from Python 2 to Python 3	2018	Python programming, Programming language evolution, Compliance	"software ecosystem" AND "empirical"
Manikas [119]	Revisiting software ecosystems Research: A longitudinal literature study	2016	Software ecosystems; Longitudinal literature study; Software ecosystem maturity	"Software ecosystems" OR "Dependency management" OR "semantic version"
Rajlich [156]	Software evolution and maintenance	2014		Software Evolution and Maintenance
Teixeira, Robles, Gonzalez-Barahona [175]	Lessons learned from applying social network analysis on an industrial Free/Libre/Open Source Software Ecosystem	2015	Social network analysis Open source Open-coopetition Software ecosystems Business models Homophily Cloud computing OpenStack	"software ecosystem analytics"

Table 4.3: Papers selected from searches using Google Scholar. The column "Query Used" describes which of the queries is used to retrieve the paper.

Author(s)	Title	Year	Keywords	Referenced In
Bavota, Canfora, Di Penta, Oliveto, Panichella [16]	How the Apache community upgrades dependencies: an evolutionary study	2014	Software Ecosystems · Project dependency upgrades · Mining software repositories	[104]
Blincoe, Harrison, Damian [30]	Ecosystems in GitHub and a method for ecosystem identification using reference coupling.	2015	Referene Coupling, Ecosystems, Technical Dependencies, GitHub, cross-reference	[45]
Cox, Bouwers, Eekelen, Visser [48]	Measuring Dependency Freshness in Software Systems	2015	software metrics, software maintenance	[96]
Decan, Mens, Claes [52]	An empirical comparison of dependency issues in OSS packaging ecosystems	2017		[3], [45], [53]
Dietrich, Jezek, Brada [58]	Broken Promises - An Empirical Study into Evolution Problems in Java Programs Caused by Library Upgrades	2014		[153]
Malloy, Power [118]	Quantifying the transition from Python 2 to 3: an empirical study of Python applications.	2017	Python, programming language evolution, language features	[117]
McDonnell, Ray, Kim [124]	An empirical study of api stability and adoption in the android ecosystem	2013		[119]

Table 4.4: Papers selected which are referenced in previously selected papers. The column "Referenced In" describes in which selected paper the paper is referenced.

Reference	Explored topic(s)	Research method(s)	Tool(s)	Dataset(s)	Ecosystem(s)	Conclusion
[3]	Use of trivial packages	Quantitative, Qualitative (Statistics over data, survey)	-	npm, GitHub	npm	Used because it is assumed to be well implemented and tested (only 45% actually has tests) and increases productivity. Quantitative research has shown that 10% of NodeJS uses trivial packages, where 16.8% are trivial packages in npm
[16]	Upgrading of dependencies	Quantitative, Qualitative (Statistics, Looking through mailing lists)	-	Apache	Apache	Upgrade done when bugfixes, but no API changes
[31]	Attitude towards breaking changes for different ecosystems	Qualitative (Interviews)	-	-	Eclipse, CRAM, npm	There are numerous ways of dealing with breaking changes and ecosystems play an essential role in the chosen way.

Reference	Explored topic(s)	Research method(s)	Tool(s)	Dataset(s)	Ecosystem(s)	Conclusion
[43]	Debian Package Incompatibilities	Quantitative	-	Debian i386 testing / stable	Debian	-
[48]	Metric for dependency freshness of a system	Qualitative / Quantitative (Statistics, Interviews)	-	Maven	Maven	Metric has been found and verified with Maven
[52]	Dependency Issues in OSS Packaging Ecosystems	Quantitative analysis (Survival analysis, statistics)	-	Eclipse, CRAM, npm	Eclipse, CRAM, npm	In all ecosystems, dependency updates result in issues, however the problems and solutions do vary.
[96]	Package dependency networks	Quantitative (Statistics)	-	npm, RubyGems, Crates.io	npm, RubyGems, Crates.io	All ecosystems are growing and over time, ecosystems become less dependent on a single popular package.
[158]	Developers response to API deprecation	Quantitative (Statistics)	-	Squeak, Pharo	Squeak, Pharo	API changes can have a large impact on ecosystem. Projects take a long time to adapt to an API change.

Reference	Explored topic(s)	Research method(s)	Tool(s)	Dataset(s)	Ecosystem(s)	Conclusion
[53]	Quantitative empirical analysis of differences and similarities between the evolution of 7 varying ecosystems	Survival analysis	-	libraries.io	Cargo, CPAN, CRAN, npm, NuGet, Packagist, RubyGems	Package updates, which may cause dependent package failures, are done on average every few months. Many packages in the analyzed package dependency networks were found to have a high number of transitive reverse dependencies, implying that package failures can affect a large number of other packages in the ecosystem.

Reference	Explored topic(s)	Research method(s)	Tool(s)	Dataset(s)	Ecosystem(s)	Conclusion
[59]	The article provides a holistic understanding of the observed and reported practices as a starting point to device specific support for the development in software ecosystems	Qualitative interview study	-	-	-	The main contribution of this article is the presentation of common features of product development and evolution in four companies. Although size, kind of software and business models differ
[87]	Code review analysis	Virtualization of process	-	Xen Github data	Xen	Analysis of code review has lead to more reviews and a more thoughtful and participatory review process. Also providing accommodations for new software developers on OSS by easy access is very important.

Table 4.5: Papers and findings for RQ1.

Reference	Explored topic(s)	Research method(s)	Dataset(s)	Ecosystem(s)	Conclusion
[84]	Exploratory study aimed at observing API evolution and its impact	Empirical study	3600 distinct systems	Pharo	After API changes, clients need time to react and rarely react at all. Replacements cannot be resolved in a uniform manner throughout the ecosystem. API changes and depreciation can present different characteristics.
[104]	An Empirical Study on the Impact of Security Advisories on Library Migration	Empirical study	4,600 GitHub software projects and 2,700 library dependencies	Github, Maven	Currently, developers do not actively update their libraries, leading to security risks.

Reference	Explored topic(s)	Research method(s)	Dataset(s)	Ecosystem(s)	Conclusion
[45]	Empirical comparison of developer retention in the RubyGems and NPM software ecosystems	Measurement of frequency and intensity of activity and retention	Github	NPM and RubyGems	Developers are more likely to abandon an ecosystem when they: 1) do not communicate with other developers, 2) do not have strong social and technical activity intensity, 3) communicate or commit less frequently and 4) do not communicate or commit for a longer period of time.
[153]	To what degree do versioning schemes provide information signals about breaking changes	Snapshot analysis	More than 100.000 jar files from Maven Central	Maven	1) Minor or major does not matter, both about 33% breaking chances, 2) breaking changes have significant impact and need fix before an upgrade, 3) Bigger libraries introduce more breaking changes, maturity is not a factor

Reference	Explored topic(s)	Research method(s)	Dataset(s)	Ecosystem(s)	Conclusion
[117]	Analysis of the transition from Python 2 to Python 3	Empirical impact study	51 applications in the Qualitas suite for Python	Python	Most Python developers choose to maintain compatibility with Python 2 and 3, thereby only using a subset of the Python 3 language and not using the new features but instead limiting themselves to a language that is no longer under active development.
[58]	Java	Empirical study into evolution problems caused by library upgrades	Qualitas corpus (Java OSS)	Java	There are currently a lot of problems caused, but some relatively minor changes to developments tools and the language could be very effective.

Table 4.6: Papers and findings for RQ2.

Reference	Open Challenges Found

Reference	Open Challenges Found
[3]	Examine relationship between team experience and project maturity and usage of trivial packages
[3]	Compare use of code snippets on Q&A sites and trivial packages
[3]	How to manage and help developers choose the best packages
[53]	Findings for one ecosystem cannot necessarily be generalized to another
[53]	Transitive dependencies are very frequent, meaning that package failures can affect a large number of other packages in the ecosystem
[88]	Determining the health of a system from an ecosystem perspective instead of project level is needed to determine which systems to use. This paper provides an initial approach but a lot more research could and should be done to determine system health.

Table 4.7: Papers and findings for RQ1.

Chapter 7

Release Engineering Analytics

7.1 Motivation

Release engineering is a software engineering discipline concerned with the development, implementation, and improvement of processes to deploy high-quality software reliably and predictably [61]. The changes made by developers of a software system should eventually be integrated and deployed such that end users may benefit from them. In recent years, release engineers have developed and adopted techniques to build infrastructures and pipelines which automate the process of releasing software to an increasingly large degree. These modern approaches have resulted in various practices such as releasing new versions of a software system in significantly shorter cycles.

Due to these developments being industry-driven, release engineering forms a largely uncharted territory for software engineering research. Efforts to close this gap would be relevant both for practitioners as well as researchers [4]. On the one hand, claims and rationales are presented by the industry to justify practices in release engineering, but these are often not empirically validated. For this reason, research should aim to build an understanding of the actual effects of release engineering practices on the software development process. On the other hand, software engineering researchers need to be aware of modern release engineering practices in order to account for them in their analyses. Otherwise, their lack of familiarity with these practices will likely result in biases in their study results.

This systematic literature review aims to provide an overview of the software analytics research that has been conducted so far on modern release engineering. Its main purpose is to identify the apparent gap between research and practice, in order to guide further research efforts.

7.1.1 Research Questions

Contrary to what is regularly the case, advances in release engineering practices are driven by industry, instead of scientific research. Building on this idea, our questions are constructed to identify in which ways existing modern release engineering practices should still be studied in software analytics research. Our review thus aims to answer the following questions.

- **RQ1:** *How is modern release engineering done in practice?*

  This question aims to identify the so-called "state of the practice" in release engineering. We will summarize practices that have been adopted to drive release engineering forward. In addition, we will identify the tools utilized to bring this about.

- **RQ2:** *What aspects of modern release engineering have been studied in software analytics research so far?*

In order to answer this question, we investigate the practices that previous empirical case studies have focused on. In doing so, we identify the associated costs and benefits that have been found, and the analysis methods used.

- **RQ3:** *What aspects of modern release engineering make for relevant study objects in future software analytics research?*

  In answering this question, we aim to identify the gap between practice and research in release engineering. This way, our intent is not only to guide but also to motivate future research.

7.2 Research Protocol

Our research has been performed following the procedures for performing systematic reviews by Kitchenham [102]. In this process, we have set up strategies for searching, selecting and quality-assessing studies. Subsequently, we have extracted data from the selected studies and synthesized the answers to our research questions.

All the papers that were found, were stored in a custom-built web-based tool for conducting literature reviews. The source code of this tool is published in a GitHub repository.[1] The tool was hosted on a virtual private server, such that all retrieved publications were stored centrally, accessible to all reviewers.

In order to save space in this chapter of the book, we have omitted the full research protocol from this chapter. The interested can find our research protocol in detail in Section 7.5.

7.3 Answers

In this section, we will give an answer to each of the research questions that we have presented in Section 7.1.1.

7.3.1 RQ1: Modern Release Engineering Practices

This section aims to answer to the question: *How is modern release engineering done in practice?*

Adams et al. [4] and Karvonen et al. [91] have described release engineering practices that are currently in use in the industry. Adams et al. [4] focused on modern release engineering, while Karvonen et al. [91] investigated agile release engineering, which is a subset of all modern release engineering. Both studies agreed that modern release engineering consists of the following components:

- **Rapid Releases (RR).**

  In contrast with traditional release cycles, RRs regularly push new releases to users in a regular schedule. As an example, FireFox releases a new version every six weeks.

- **DevOps.**

  According to Dyck et al. [61]: "DevOps is an organizational approach that stresses empathy and cross-functional collaboration within and between teams (especially development and IT operations) in software development organizations, in order to operate resilient systems and accelerate delivery of changes."

- **Continuous Integration (CI).**

[1] See https://github.com/jessetilro/research

To quote Adams et al. [4]: "[CI] refers to the activity of continuously polling the [version control system] for new commits or merges, checking these revisions out on dedicated build machines, compiling them and running an initial set of tests to check for regressions."

- **Continuous Deployment or Continuous Delivery.**

  When the CI tests pass, the code can be automatically deployed to the production environment. The difference between these terms is that continuous delivery does not require that changes are automatically deployed, but continuous deployment always automatically deploys changes. However, the change might not yet be released, because it can be hidden using feature toggles [106].

Besides these four components, Adams et al. [4] also identified three other concepts that are used in modern release engineering, specifically to release software as often as possible and thus enable continuous deployment or delivery:

- **Branching and Merging.**

  There are several possible strategies of branching and merging. Typically, merging must be done as often as possible in order to release as often as possible.

- **Build System.**

  Building the software must be done in a consistent way, such that each build produces the same result. With the build configuration stored inside the project, every developer (or automated tool) only needs to issue a single command in order to build the project, instead of manually having to configure the build process every time.

- **Infrastructure-as-Code.**

  In the same alley of "storing configuration", infrastructure-as-code means that the server (or virtual machine) on which the software product is running can also be automatically configured with code, instead of having to configure each server manually.

Besides these seven components that are more technical, Poo-Caamaño [151] has identified that there are also social aspects to modern release engineering. Specifically, most large software projects have a dedicated Release Team that will decide on the release strategies and communicate them to others.

7.3.2 RQ2: Studied Parts of Release Engineering

This section aims to answer to the question: *What aspects of modern release engineering have been studied in software analytics research so far?*

Over the years, the software industry has come up with inventive approaches to deliver new features and fixes in a more efficient and faster manner. This has resulted in case studies being done to assess the associated risk and cost factors, and what benefits certain strategies can give.

Khomh et al. [94] have looked into the effects of switching from traditional to rapid release cycles in the case of Mozilla Firefox. The paper has concluded that users do not experience significantly more post-release bugs, bugs are fixed faster, but that users experience bugs earlier in the software execution. Mantyla et al. [123] have also considered data from Mozilla Firefox and has examined the impact of release engineering on testing efforts. Observations of the paper conclude that the rapid release cycle performs more test executions per day, but these tests focus on a smaller subset of the test case corpus and that testing happens closer to release and is more continuous. A limitation of this study is that it measures correlation, rather than causation. Da Costa et al. [46] has further zoomed into the integration of addressed issues and has considered data from Mozilla Firefox, as well as data from ArgoUML and Eclipse. The paper found that addressed issues are usually delayed in a rapid release cycle and are often excluded from releases. Similar conclusions based on Mozilla Firefox were made by Da Costa et al. [47], who found that minor-traditional releases tend to have less integration delay than major/minor-rapid releases.

Castelluccio et al. [35] has examined the practice of *patch uplifting* in the release management at Mozilla Firefox where patches that fix critical issues, or implement high-value features are often promoted directly from the development channel to a stabilization channel. The paper evaluated the characteristics of patch uplift decisions and interviewed three Mozilla release managers. The paper concluded that the majority of patch uplift decisions are made due to a wrong functionality or crash. The specificity and code author of patches that are requested to be uplifted are also a major factor for release managers.

In response to case studies being done on many prominent open source software projects, Teixeira [174] has described OpenStack's shift to a liberal six-month release cycle. As this is an ongoing study, the results given by the paper are preliminary and only observe the process. OpenStack's release process can be considered as a hybrid of feature-based and time-based releases. OpenStack encourages regular releases but also attempts to include new features at each regular release.

Rather than focussing on topics such as issue and delays, Poo-Caamaño [151] focusses on the communication in release engineering in the cases of GNOME and OpenStack. Through analyzing over 2.5 years of communication, the paper has made a number of observations. The paper found that developers tend to communicate through blogs, bug trackers, conferences, and hackfests. Another finding is that a release team is set to define requirements, quality standards, and coordination through (direct) communication. Although only the mailing lists of the projects were studied, defined challenges include keeping everyone informed and engaged, monitoring changes and setting priorities in cross-project coordination.

Laukkanen et al. [106] have described what effects modern release engineering have on software with different organizational contexts. This study specifically focusses on continuous deployment practices. The paper has found that high internal quality standards combined with the large distributed organizational context of large corporations slowed the verification process down and therefore had a negative impact on release capability. However, in small corporations, the lack of internal verification measures due to a lack of resources was mitigated by code review, disciplined CI and external verification by customers in customer environments. More about the factors that can play a role is addressed by Rodríguez et al. [160], where an overview of contributing factors in continuous deployment are defined and categorized based on literature between 2001 and 2014.

As rapid release cycles and continuous deployment are topics that are new and emerging, not enough research has been done to generalize any conclusions that are made in the case studies discussed in this section. This is why all the empirical studies in this survey have one major sidenote in common: more case studies are needed. Open challenges such as these will be discussed in the next section.

7.3.3 RQ3: Future Research

This section aims to answer the question: *What aspects of modern release engineering make for relevant study objects in future software analytics research?*

7.3.3.1 General Suggestions

The body of literature that we analyzed for this survey mostly comprised case studies that employed quantitative analysis methods. From these studies, interesting conclusions have been drawn about the effects of release engineering practices on software development processes in specific contexts. However, the generalizability of the findings in these case studies is very limited. Therefore, in general many studies suggest that future research efforts focus on performing additional case studies, both to verify existing findings and to study new relationships and new contexts [4, 35, 42, 91, 94, 106, 174]. It also seems worthwhile to triangulate findings by complementing data analyses with other quantitative (e.g. a survey) or qualitative (e.g. an interview) methods [91]. Finally, additional literature reviews will allow researchers to keep an overview of the most recent developments and findings in the area of release engineering [106, 160].

Apart from verifying results, it might be worthwhile to leverage them by constructing analysis tools for practitioners. For example, Castelluccio et al. [35] suggest exploring possibilities to leverage their research

by building classifiers capable of automatically assessing the risk associated with patch uplift candidates and recommend patches that can be uplifted safely. Also, companies seem to be struggling with the adoption of continuous delivery and deployment, so a checklist for analyzing readiness for these practices might be developed [91].

The review by Karvonen et al. [91] makes a number of general suggestions for future research. In particular, there should be more attention to comprehensively reporting how practices are implemented and in which context they are embedded, instead of just stating that they are used. Also, the viewpoints of different stakeholders other than developers can be taken into account. For example, the customer perceptions regarding the adoption of a certain practice can be investigated.

7.3.3.2 Directions for Specific Practices

Rapid Releases

As established, rapid releases are a prevalent topic in current research on modern release engineering. However, it will be useful to verify these results given the fact that this research mainly involves case studies (most of which are only concerned with Mozilla Firefox due to the availability of data). To this end, there are opportunities to further investigate the effects of switching to rapid releases on:

- code integration [35, 46, 47, 169],
- testing efforts [123],
- software quality [94, 95],
- (library) adoption [67]
- time pressure and work patterns [42].

DevOps

When it comes to DevOps, future research is needed to refine its definition such that it is uniform and valid for many situations [61]. According to Karvonen et al. [91], it seems that the goals in DevOps are congruent with those in release engineering, and future research on this topic is therefore highly relevant in order to study modern release engineering.

Continuous Delivery / Deployment

Research on continuous deployment seems to be still in its infancy, therefore Rodríguez et al. [160] have suggested a significant number of different concrete opportunities for future research. In general, they conclude that the topic needs an increase in the number and rigor of empirical studies, and thus it presents opportunities for software analytics research. In a systematic literature review, Laukkanen et al. [105] identified 40 problems, 28 causal relationships and 29 solutions related to the adoption of continuous delivery. These problems and solutions can be studied further to deepen the understanding of the nature of these problems and how to apply their solutions. Some of the problems that are more of a human or organizational nature might be involved with a broader spectrum of changes, so it should also be investigated to what extent these problems are specific to continuous delivery.

Continuous Integration / Build system

One of the main issues that seems to be obstructing organizations in adopting modern release engineering practices is build design [105]. In a case study by Laukkanen et al. [106], it was found that a complex automated build and integration system led to a more undisciplined one, which in turn slowed down the verification and release processes. Therefore, future research might investigate how developers can make their builds more maintainable and of higher quality, how anti-patterns in the design of the build can be refactored, and how continuous integration can be made faster and more energy efficient [4].

7.4 Conclusion

In this literature survey, we have provided an answer to the following three research questions:

- **RQ1:** *How is modern release engineering done in practice?*

  We found that there are six important technical aspects to modern release engineering: Rapid Releases, DevOps, Continuous Integration, Continuous Deployment, Branching and Merging, Build Configuration and Infrastructure-as-code. The most important social aspect of modern release engineering is communication.

- **RQ2:** *What aspects of modern release engineering have been studied in software analytics research so far?*

  At this point in time, case studies have mainly focussed on the resulting factors of switching from a traditional release cycle to a rapid release cycle, and what effects this has in various organizational contexts. As all included studies suggest, more empirical studies are needed to be able to make general conclusions in the novel field of release engineering.

- **RQ3:** *What aspects of modern release engineering make for relevant study objects in future software analytics research?*

  In general, more empirical research is required to validate and generalize the results of many previous case studies within the field of release engineering. In addition to performing case studies involving quantitative analyses, it may be beneficial to triangulate results using various research methods. Also, future research should more comprehensively describe how practices are implemented, and consider different stakeholders. For each practice, future research is suggested on the one hand to further investigate their effects on the development process, and on the other hand to investigate problems involved with their adoption.

7.5 Appendix

This appendix contains sections that were part of our process during this literature survey. They are not directly needed to answer the research questions, but are still relevant in order to validate our survey. This Appendix contains our project timetable, the research protocol in full detail, and the raw extracted data from the selected studies.

7.5.1 Project Timetable

The literature review was conducted over the course of four weeks. We worked iteratively and planned for four weekly milestones.

Milestone	Deadline	Goals
1	16/9/18	• Develop the search strategy • Collect initial publications
2	23/9/18	• Write full research protocol

Milestone	Deadline	Goals
3	30/9/18	• Collect additional literature according to the protocol • Perform data extraction
4	7/10/18	• Perform data synthesis • Write final version of the chapter

7.5.2 Research Protocol

In this appendix, we will describe in detail how we applied the protocol for performing systematic literature reviews by Kitchenham [102]. In order, we will go over the search strategy, study selection, study quality assessment, and data extraction. The last subsection will list which studies we included in this review and which we have found, but excluded from the review for a specific reason.

7.5.2.1 Search Strategy

Since release engineering is a relatively new research topic, we took an exploratory approach in collecting any literature revolving around the topic of release engineering from the perspective of software analytics. This aided us to determine a more narrow scope for our survey, subsequently to allow us to find additional literature to fit this scope.

At the start of this project, we were provided with an initial seed of five papers as a starting point for our literature survey [4, 46, 47, 94, 95].

We collected other publications using two search engines: Scopus and Google Scholar. Each of the two search engines comprises several databases such as ACM Digital Library, Springer, IEEE Xplore and ScienceDirect. The main query that we constructed is displayed in Figure 1. The publications found using this query were:

- [92]
- [93]
- [35]
- [91]
- [42]
- [67]
- [169]
- [106]
- [61]

```
TITLE-ABS-KEY(
  (
    "continuous release" OR "rapid release" OR "frequent release"
    OR "quick release" OR "speedy release" OR "accelerated release"
    OR "agile release" OR "short release" OR "shorter release"
    OR "lightning release" OR "brisk release" OR "hasty release"
    OR "compressed release" OR "release length" OR "release size"
    OR "release cadence" OR "release frequency"
    OR "continuous delivery" OR "rapid delivery" OR "frequent delivery"
    OR "fast delivery" OR "quick delivery" OR "speedy delivery"
```

```
    OR "accelerated delivery" OR "agile delivery" OR "short delivery"
    OR "lightning delivery" OR "brisk delivery" OR "hasty delivery"
    OR "compressed delivery" OR "delivery length" OR "delivery size"
    OR "delivery cadence" OR "continuous deployment" OR "rapid deployment"
    OR "frequent deployment" OR "fast deployment" OR "quick deployment"
    OR "speedy deployment" OR "accelerated deployment" OR "agile deployment"
    OR "short deployment" OR "lightning deployment" OR "brisk deployment"
    OR "hasty deployment" OR "compressed deployment" OR "deployment length"
    OR "deployment size" OR "deployment cadence"
  ) AND (
    "release schedule" OR "release management" OR "release engineering"
    OR "release cycle" OR "release pipeline" OR "release process"
    OR "release model" OR "release strategy" OR "release strategies"
    OR "release infrastructure"
  )
  AND software
) AND (
  LIMIT-TO(SUBJAREA, "COMP") OR LIMIT-TO(SUBJAREA, "ENGI")
)
AND PUBYEAR AFT 2014
```

Figure 1. Query used for retrieving release engineering publications via Scopus.

In addition to querying search engines as described above, references related to retrieved papers were analyzed. These reference lists were obtained from Google Scholar and from the *References* section in the papers themselves. We selected all papers on release engineering that are citing or being cited by the initial set of papers. Using this approach, we have found six additional papers. The results of the reference analysis are listed in Table 1.

Table 1. Papers found indirectly by investigating citations of/by other papers.

Starting point	Type	Result
[169]	has cited	[150] [123]
[94]	is cited by	[151] [174]
[123]	is cited by	[160] [39]
[106]	has cited	[105]

All the papers that were found, were stored in a custom-built web-based tool for conducting literature reviews. The source code of this tool is published in a GitHub repository.[2] The tool was hosted on a virtual private server, such that all retrieved publications were stored centrally, accessible to all reviewers.

7.5.2.2 Study Selection

We selected the studies that we wanted to include in the survey with aid of the aforementioned tool for storing the papers. In this tool, it is possible to label papers with tags and leave comments and ratings. Every paper is reviewed based on the inclusion and exclusion criteria. Based on this, the tool allowed to filter out all papers that appeared not to be relevant for this literature survey.

We only used one exclusion criteria: studies that are published before 2014, will not be included in our survey (this is enforced by our search query). The inclusion criteria are as follows:

- The study must show (at least) one release engineering technique.

[2] See https://github.com/jessetilro/research

- The study must not just show a release engineering technique, but analyze its performance compared to other techniques.

The last subsection of this appendix lists which studies were selected and which were discareded.

7.5.2.3 Study Quality Assessment

Based on Kitchenham [102], the quality of a paper will be assessed by the evidence it provides, based on the following scale. All levels of quality in this scale will be accepted, except for level 5 (evidence obtained from expert opinion).

1. Evidence obtained from at least one properly-designed randomised controlled trial.
2. Evidence obtained from well-designed pseudo-randomised controlled trials (i.e. non-random allocation to treatment).
3. Comparative studies in a real-world setting:
    1. Evidence obtained from comparative studies with concurrent controls and allocation not randomised, cohort studies, case-control studies or interrupted time series with a control group.
    2. Evidence obtained from comparative studies with historical control, two or more single arm studies, or interrupted time series without a parallel control group.
4. Experiments in artificial settings:
    1. Evidence obtained from a randomised experiment performed in an artificial setting.
    2. Evidence obtained from case series, either post-test or pre-test/post-test.
    3. Evidence obtained from a quasi-random experiment performed in an artificial setting.
5. Evidence obtained from expert opinion based on theory or consensus.

Also, the studies will be examined to see if they contain any type of bias. For this, the same types of biases will be used as described by Kitchenham@kitchenham2004procedures:

- Selection/Allocation bias: Systematic difference between comparison groups with respect to treatment.
- Performance bias: Systematic difference is the conduct of comparison groups apart from the treatment being evaluated.
- Measurement/Detection bias: Systematic difference between the groups in how outcomes are ascertained.
- Attrition/Exclusion bias: Systematic differences between comparison groups in terms of withdrawals or exclusions of participants from the study sample.

The studies will be labeled by their quality level and possible biases. This information can be used during the Data Synthesis phase to weigh the importance of individual studies [102].

7.5.2.4 Data Extraction

To accurately capture the information contributed by each publication in our survey, we will use a systematic approach to extracting data. To guide this process, we will be using a data extraction form which describes what aspects of a publication are crucial to record. Besides general publication information (title, author etc.), the form contains questions that are based on our defined research questions. Furthermore, the form contains a section for quantitative research, where aspects such as population and evaluation will be documented. The form that is used for this is shown below:

```
General information:

- Name of person extracting data:
- Date form completed (dd/mm/yyyy):
- Publication title:
- Author information:
- Publication type:
- Conference/Journal:
```

```
- Type of study:

What practices in release engineering does this publication mention?

Are these practices to be classified under dated, state of the art or state of
the practice? Why?

What open challenges in release engineering does this publication mention?

What research gaps does this publication contain?

Are these research gaps filled by any other publications in this survey?

Quantitative research publications:

- Study start date:
- Study end date or duration:
- Population description:
- Method(s) of recruitment of participants:
- Sample size:
- Evaluation/measurement description:
- Outcomes:
- Limitations:
- Future research:

Notes:
```

7.5.2.5 Data Synthesis

To summarize the contributions and limitations of each of the included publications, we will apply a descriptive synthesis approach. In this part of our survey, we will compare the data that was extracted of the included publications. Publications with similar findings will be grouped and evaluated, and differences between groups of publications will be structured and elaborated on. In this we will compare them using specifics such as their study types, time of publication and study quality.

If the extracted data allows for a structured tabular visualization of similarities and differences between publications this we serve as an additional form of synthesis. However, this depends on the final included publications of this survey.

7.5.2.6 Included and Excluded Studies

Included:

- [4]
- [35]
- [39]
- [42]
- [46]
- [47]
- [61]
- [67]
- [91]
- [93]
- [94]

- [105]
- [106]
- [123]
- [150]
- [151]
- [160]
- [169]
- [174]

Excluded:

- [95] has been excluded, because it presents the same results as [94], while the latter is more extensive because it is a journal article instead of a conference article.
- [92] has been excluded, because we could not obtain the actual paper since it has not yet been officially released.

7.5.3 Raw Extracted Data

7.5.3.1 Understanding the impact of rapid releases on software quality – The Case of Firefox

Reference: [94]

General information:

- Name of person extracting data: Maarten Sijm
- Date form completed: 27-09-2018
- Author information: Foutse Khomh, Bram Adams, Tejinder Dhaliwal, Ying Zou
- Publication type: Paper in Conference Proceedings
- Conference: Mining Software Repositories (MSR)
- Type of study: Quantitative, empirical case study

What practices in release engineering does this publication mention?

- Changing from traditional to rapid release cycles in Mozilla Firefox

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice, because they study Firefox and Firefox is still using rapid release cycles. However, it is dated because the data is six years old.

What open challenges in release engineering does this publication mention?

- More case studies are needed

What research gaps does this publication contain?

- More case studies are needed

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: 01-01-2010 (Firefox 3.6)
- Study end date or duration: 20-12-2011 (Firefox 9.0)
- Population description: Mozilla Wiki, VCS, Crash Repository, Bug Repository
- Method(s) of recruitment of participants: N/A (case study)
- Sample size: 25 alpha versions, 25 beta versions, 29 minor versions and 7 major versions. Amount of bugs/commits/etc. is not specified.
- Evaluation/measurement description: Wilcoxon rank sum test

- Outcomes:
  - With shorter release cycles, users do not experience significantly more post-release bugs
  - Bugs are fixed faster
  - Users experience these bugs earlier during software execution (the program crashes earlier)
- Limitations: Results are specific to Firefox
- Future research: More case studies are needed

7.5.3.2 On the influence of release engineering on software reputation

Reference: [150]

General information:

- Name of person extracting data: Maarten Sijm
- Date form completed: 27-09-2018
- Author information: Christian Plewnia, Andrej Dyck, Horst Lichter
- Publication type: Paper in Conference Proceedings
- Conference: 2nd International Workshop on Release Engineering
- Type of study: Quantitative, empirical case study on multiple software

What practices in release engineering does this publication mention?

- Rapid releases

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- Dated practice, data is from before 2014

What open challenges in release engineering does this publication mention?

- Identifying software reputation can better be done using a qualitative study.

What research gaps does this publication contain?

- Identifying software reputation can better be done using a qualitative study.

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: Q3 2008
- Study end date or duration: Q4 2013
- Population description: Chrome, Firefox, Internet Explorer
- Method(s) of recruitment of participants: N/A (case study)
- Sample size: 3 browsers
- Evaluation/measurement description: No statistical analysis, just presenting market share results
- Outcomes:
  - Chrome's market share increased after adopting rapid releases
  - Firefox's market share decreased after adopting rapid releases
  - IE's market share decreased
- Limitations:
  - Identifying software reputation can better be done using a qualitative study.
- Future research:
  - Identifying software reputation can better be done using a qualitative study.

7.5.3.3 On rapid releases and software testing: a case study and a semi-systematic literature review

Reference: [123]

General information:

- Name of person extracting data: Maarten Sijm
- Date form completed: 28-09-2018
- Author information: Mäntylä, Mika V. and Adams, Bram and Khomh, Foutse and Engström, Emelie and Petersen, Kai
- Publication type: Journal/Magazine Article
- Journal: Empirical Software Engineering
- Type of study: Empirical case study and semi-systematic literature review

What practices in release engineering does this publication mention?

- Impact of rapid releases on testing effort

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice for the case study
- State of the art for the literature review

What open challenges in release engineering does this publication mention?

- Future work should focus on empirical studies of these factors that complement the existing qualitative observations and perceptions of rapid releases.

What research gaps does this publication contain?

- See open challenges

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: June 2006 (Firefox 2.0)
- Study end date or duration: June 2012 (Firefox 13.0)
- Population description: System-level test execution data
- Method(s) of recruitment of participants: N/A (case study)
- Sample size: 1,547 unique test cases, 312,502 executions, performed by 6,058 individuals on 2,009 software builds, 22 OS versions and 78 locales.
- Evaluation/measurement description: Wilcoxon rank-sum test, Cliff's delta, Cohen's Kappa for Firefox Research Question (FF-RQ) 5.
- Outcomes (FF-RQs; RR = rapid release; TR = traditional release):
  1. RRs perform more test executions per day, but these tests focus on a smaller subset of the test case corpus.
  2. RRs have less testers, but they have a higher workload.
  3. RRs test fewer, but larger builds.
  4. RRs test fewer platforms in total, but test each supported platform more thoroughly.
  5. RRs have higher similarity of test suites and testers within a release series than TRs had.
  6. RR testing happens closer to the release date and is more continuous, yet these findings were not confirmed by the QA engineer.
- Limitations:
  - Study measures correlation, not causation
  - Not generalizable, as it is a case study on FF
- Future research: More empirical studies

Semi-systematic literature survey:

- Study date: Unknown (before 2015)
- Population description: Papers with main focus on:
  - Rapid Releases (RRs)
  - Aspect of software engineering largely impacted by RRs
  - An agile, lean or open source process having results of RRs
  - Excluding: opinion papers without empirical data on RRs
- Method(s) of recruitment of participants: Scopus queries
- Sample size: 24 papers
- Outcomes:
  - Evidence is scarce. Often RRs are implemented as part of agile adoption. This makes it difficult to separate the impact of RRs from other process changes.
  - Originates from several software development paradigms: Agile, FOSS, Lean, internet-speed software development
  - Prevalence
    - Practiced in many software engineering domains, not just web applications
    - Between 23% and 83% of practitioners do RRs
  - (Perceived) Problems:
    - Increased technical debt
    - RRs are in conflict with high reliability and high test coverage
    - Customers might be dipleased with RRs (many updates)
    - Time-pressure / Deadline oriented work
  - (Perceived) Benefits:
    - Rapid feedback leading to increased quality focus of the devs and testers
    - Easier monitoring of progress and quality
    - Customer satisfaction
    - Shorter time-to-market
    - Continuous work / testing
  - Enablers:
    - Sequential development where multiple releases are under work simultaneously
    - Tools for automated testing and efficient deployment
    - Involvement of product management and productive customers
- Limitations:
  - Not all papers that present results about RRs, have “rapid release” mentioned in the abstract.
- Future research:
  - Systematically search for agile and lean adoption papers

7.5.3.4 Release management in free and open source software ecosystems

Reference: [151]

General information:

- Name of person extracting data: Maarten Sijm
- Date form completed: 28-09-2018
- Author information: Germán Poo-Caamaño
- Publication type: PhD Thesis
- Type of study: Empirical case study on two large-scale FOSSs: GNOME and OpenStack

What practices in release engineering does this publication mention?

- Communication in release engineering

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice, because case study

What open challenges in release engineering does this publication mention?

- Is the ecosystem [around the studied software] shrinking or expanding?
- How have communications in the ecosystem changed over time?

What research gaps does this publication contain?

- More case studies are needed

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications (GNOME):

- Study start date: January 2009 (GNOME 2.x)
- Study end date or duration: August 2011 (GNOME 3.x)
- Population description: Mailing lists
- Method(s) of recruitment of participants: GNOME's website recommends this channel of communication. IRC is also recommended, but its history is not stored.
- Sample size: 285 mailing lists, 6947 messages, grouped into 945 discussions.
- Evaluation/measurement description: Counting
- Outcomes:
  - Developers also communicate via blogs, bug trackers, conferences, and hackfests.
  - The Release Team has direct contact with almost all participants in the mailing list
  - The tasks of the Release Team:
    - defining requirements of GNOME releases
    - coordinating and communicating with projects and teams
    - shipping a release within defined quality and time specifications
  - Major challenges of the Release Team:
    - coordinate projects and teams of volunteers without direct power over them
    - keep the build process manageable
    - monitor for unplanned changes
    - monitor for changes during the stabilization phase
    - test the GNOME release
- Limitations:
  - Only mailing list was investigated, other channels were not
  - Possible subjective bias in manually categorizing email subjects
  - Not very generalizable, as it's just one case study
- Future research:
  - Fix the limitations

Quantitative research publications (OpenStack):

- Study start date: May 2012
- Study end date or duration: July 2014
- Population description: Mailing lists
- Method(s) of recruitment of participants: Found on OpenStack's website
- Sample size: 47 mailing lists, 24,643 messages, grouped into 7,650 discussions. Filtered data: 14,486 messages grouped into 2,682 discussions.
- Evaluation/measurement description: Counting
- Outcomes:
  - Developers communicate via email, blogs, launchpad, wiki, gerrit, face-to-face, IRC, video-conferences, and etherpad.
  - Project Team Leaders and the Release Team members are the key players in the communication and coordination across projects in the context of release management
  - The tasks for the Release Team and Project Team Leaders:
    - defining the requirements of an OpenStack release

    - * coordinating and communicating with projects and teams to reach the objectives of each milestone
    - * coordinating feature freeze exceptions at the end of a release
    - * shipping a release within defined quality and time specifications
  - Major challenges of these teams:
    - * coordinate projects and teams without direct power over them
    - * keep everyone informed and engaged
    - * decide what becomes part of of the integrated release
    - * monitor changes
    - * set priorities in cross-project coordination
    - * overcome limitations of the communication infrastructure
- Limitations:
  - Only studies mailing list, to compare with GNOME case study
  - Possible subjective bias in manually categorizing email subjects
  - Not very generalizable, as it's just one case study
- Future research:
  - Fix the limitations

Notes:

- Since there are two case studies, the results become a bit more generalizable
- The author set up a theory that encapsulates the communication and coordination regarding release management in FOSS ecosystems, and can be summarized as:
  1. The size and complexity of the integrated product is constrained by the release managers capacity
  2. The release management should reach the whole ecosystem to increase awareness and participation
  3. The release managers need social and technical skills

7.5.3.5 Release Early, Release Often and Release on Time. An Empirical Case Study of Release Management

Reference: [174]

General information:

- Name of person extracting data: Maarten Sijm
- Date form completed: 28-09-2018
- Author information: Jose Teixeira
- Publication type: Paper in Conference Proceedings
- Conference: Open Source Systems: Towards Robust Practices
- Type of study: Empirical case study

What practices in release engineering does this publication mention?

- Shifting towards rapid releases in OpenStack

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice, because it is a recent case study on OpenStack

What open challenges in release engineering does this publication mention?

- More case studies are needed.

What research gaps does this publication contain?

- More case studies are needed.

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: Not specified
- Study end date or duration: Not specified
- Population description: Websites and blogs
- Method(s) of recruitment of participants: Random clicking through OpenStack websites
- Sample size: Not specified
- Evaluation/measurement description: Not specified
- Outcomes:
  - OpenStack releases in a cycle of six months
  - The release management process is a hybrid of feature-based and time-based
  - Having a time-based release strategy is a challenging coopearative task involving multiple people and technology
- Limitations:
  - Study is not completed yet, these are preliminary results
- Future research:
  - Not indicated

7.5.3.6 Kanbanize the release engineering process

Reference: [93]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Kerzazi, N. and Robillard, P.N.
- Publication type: Paper in Conference Proceedings
- Journal: 2013 1st International Workshop on Release Engineering, RELENG 2013 - Proceedings
- Type of study: Action research

What practices in release engineering does this publication mention?

- Following principles of the Kanban agile software development life-cycle model that implicitly describe the release process
- (Switching to) more frequent (daily) release cycles
- (Transitioning to) a structured release process

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- Either dated or state of the practice, not sure. Would have to do some additional research on the adoption of Kanban

What open challenges in release engineering does this publication mention?

- Release effectiveness: minimize system failure and customer impact
- Problems with releasing encountered in practice

What research gaps does this publication contain?

-

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date:
- Study end date or duration:
- Population description:

- Method(s) of recruitment of participants:
- Sample size:
- Evaluation/measurement description:
- Outcomes:
  1.
- Limitations:
- Future research:

Notes:

-

7.5.3.7 Is it safe to uplift this patch? An empirical study on mozilla firefox

Reference: [35]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Castelluccio, M. and An, L. and Khomh, F.
- Publication type: Paper in Conference Proceedings
- Journal: Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017
- Type of study: Case study, both quantitative (data analysis) and qualitative (interviews)

What practices in release engineering does this publication mention?

- Patch uplift (meaning the promotion of patches from development directly to a stabilization channel, potentially skipping several channels)

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice: case study of what is being done in the field, quite recently (2017).

What open challenges in release engineering does this publication mention?

- Exploring possibilities to leverage this research by building classifiers capable of automatically assessing the risk associated with patch uplift candidates and recommend patches that can be uplifted safely.
- Validate and extend results of this study for generalizability.

What research gaps does this publication contain?

- Study aimed to fill two identified gaps identified in literature:
  - How do urgent patches in rapid release models affect software quality (in terms of fault proneness)?
  - How can the reliability of the integration of urgent patches be improved?

Are these research gaps filled by any other publications in this survey?

- The paper itself

Quantitative research publications:

- Study start date:
- Study end date or duration:
- Population description:
- Method(s) of recruitment of participants:
- Sample size:
- Evaluation/measurement description:
- Outcomes:
  1.

- Limitations:
- Future research:

Notes:

-

7.5.3.8 Systematic literature review on the impacts of agile release engineering practices

Reference: [91]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Karvonen, T. and Behutiye, W. and Oivo, M. and Kuvaja, P.
- Publication type: Journal/Magazine Article
- Journal: Information and Software Technology
- Type of study: Systematic literature review

What practices in release engineering does this publication mention?

- Agile release engineering (ARE) practices
  - Continuous integration (CI)
  - Continuous delivery (CD)
  - Rapid Release (RR)
  - Continuous deployment
  - DevOps (similar to CD, congruent with release engineering practices)

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the art, for it concerns a state of the art report and was published recently (2017).

What open challenges in release engineering does this publication mention?

- Claims that modern release engineering practices allow for software to be delivered faster and cheaper should be further empirically validated.
- This analysis could be extended with industry case studies, to develop a checklist for analyzing company and ecosystem readiness for continuous delivery and continuous deployment.
- The comprehensive reporting of the context and how the practice is implemented instead of merely referring to usage of the practice should be considered by future research.
- Different stakeholders' points of view, such as customer perceptions regarding practices require further research.
- Research on DevOps would be highly relevant for release engineering and the continuous software engineering research domain.
- Future research on the impact of RE practices could benefit from more extensive use of quantitative methodologies from case studies, and the combination of quantitative with qualitative (e.g. interviews) methods.

What research gaps does this publication contain?

- Refer to challenges

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: N/A
- Study end date or duration: N/A

- Population description: N/A
- Method(s) of recruitment of participants: N/A
- Sample size: N/A
- Evaluation/measurement description: N/A
- Outcomes: N/A
- Limitations: N/A
- Future research: N/A

Notes:

-

7.5.3.9 Abnormal Working Hours: Effect of Rapid Releases and Implications to Work Content

Reference: [42]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Claes, M. and Mantyla, M. and Kuutila, M. and Adams, B.
- Publication type: Paper in Conference Proceedings
- Journal: IEEE International Working Conference on Mining Software Repositories
- Type of study: Quantitative case study

What practices in release engineering does this publication mention?

- Faster release cycles

Are these practices to be classified under dated, state of the art or state of the practice? Why?

-

What open challenges in release engineering does this publication mention?

- Future research might further study the impact of time pressure and work patterns - indirectly release practices - on software developers.

What research gaps does this publication contain?

-

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: first data item 2012-12-21
- Study end date or duration: last data item 2016-01-03
- Population description: N/A
- Method(s) of recruitment of participants: N/A
- Sample size: 145691 bug tracker contributors (1.8% timezone), 11.11 million comments (53% author with timezone)
- Evaluation/measurement description: measure distributions on number of comments per day of the week and time of the day, before and after transition to rapid release cycles. Test distribution difference using Mann-Whitney U test and test effect size using Cohen's d and Cliff's delta. Also evaluate general development of number of comments, working day against weekend and day against night.
- Outcomes:
  1. Switching to rapid releases has reduced the amount of work performed outside of office hours. (Supported by results in psychology.)

2. Thus, rapid release cycles seem to have a positive effect on occupational health.
3. Comments posted during the weekend contained more technical terms.
4. Comments posted during weekdays contained more positive and polite vocabulary.

- Limitations:
- Future research:

Notes:

-

7.5.3.10 Does the release cycle of a library project influence when it is adopted by a client project?

Reference: [67]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Fujibayashi, D. and Ihara, A. and Suwa, H. and Kula, R.G. and Matsumoto, K.
- Publication type: Paper in Conference Proceedings
- Journal: SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering
- Type of study: Quantitative study

What practices in release engineering does this publication mention?

- Rapid release cycles

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the art and practice: practitioners currently practice it, researchers currently research it.

What open challenges in release engineering does this publication mention?

- Gaining an understanding of the effect of a library's release cycle on its adoption.

What research gaps does this publication contain?

- First step towards solving the above challenge.

Are these research gaps filled by any other publications in this survey?

- This paper

Quantitative research publications:

- Study start date: 21-07-2016 (data extraction)
- Study end date or duration:
- Population description:
- Method(s) of recruitment of participants:
- Sample size: 23 libraries, 415 client projects
- Evaluation/measurement description:
- Scott-Knott test to group libraries with similar release cycle.
- Outcomes:
  1. There is a relationship between release cycle of a library project and the time for clients to adopt it: quicker release seems to be associated with quicker adoption.
- Limitations:
  - Small sample size
  - Not controlled for many factors
  - No statistical significance tests?

- Future research:

Notes:

- Very short, probably not very strong evidence, refer to limitations
- Nice that the focus is libraries here, very interesting population because most studies focus on end-user targeting software systems

7.5.3.11 Rapid releases and patch backouts: A software analytics approach

Reference: [169]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Souza, R. and Chavez, C. and Bittencourt, R.A.
- Publication type: Journal/Magazine Article
- Journal: IEEE Software
- Type of study: Quantitative case study (Mozilla Firefox)

What practices in release engineering does this publication mention?

- Rapid release
- Backing out of broken patches (patch backouts)
- Stabilization channels / monitored integration repository

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice (case study)

What open challenges in release engineering does this publication mention?

- How rapid release cycles affect code integration, where patch backouts are a proxy for studying code integration
- Integrate backout rate analysis in an analytics tool to provide release engineers with up-to-date information on the process

Quantitative research publications:

- Study start date: first data item 30 june 2009
- Study end date or duration: last data item 17 september 2013
- Population description:
- Method(s) of recruitment of participants:
- Sample size: 43198 bug fixes, no further sample sizes of the raw data mentioned anywhere unfortunately. (Data from Mozilla Firefox project.)
- Evaluation/measurement description: Associate commit log, bug reports and releases. Classify backouts. Measure rate of backouts against all fixed bugs, per month and per release strategy period. Test for statistical significance using Fisher's exact test and Wilcoxon signed-rank test.
- Outcomes:
  1. Absolute numbers of bug fixes and backouts increased under rapid releases (probably the increase in regular contributors played a role, cannot conclude anything about workload.)
  2. Backout rate increased under rapid releases (sheriff managed integration repositories may have increased the prevalence of backout culture)
  3. Higher early backout rate and lower late backout rate indicate a shift towards earlier problem detection (proportion early from 57 to 88 %) The time-to-backout also dropped.
- Limitations:
  - Sample size not mentioned
  - Quite trivial statistics

- Future research:
  - Integrate backout rate analysis in an analytics tool to provide release engineers with up-to-date information on the process

Interview triangulation

- Explanations of quantitative outcomes:
  - larger code base and more products -> more conflicts
  - evolution of automated testing toolset -> earlier and more backouts
  - sheriff managed integration repos -> earlier and more backouts
- Explanations of impact
  - cultural shift reduced testing efforts beforehand, and higher early backout rate eventually reduced the effort to integrate bug fixes for developers
  - given the many stabilization channels and the rarity of very late backouts both in traditional and rapid release cycles, changes in backouts do not seem to influence users' perception of quality (even though frequent update notifications and broken compatibilities caused upset users)

Notes:

- Also reviews existing literature well.
- Treats transitional period from traditional to rapid releases as a separate period.

7.5.3.12 Comparison of release engineering practices in a large mature company and a startup

Reference: [106]

General information:

- Name of person extracting data: Jesse Tilro
- Date form completed: 29-09-2018
- Author information: Laukkanen, E. and Paasivaara, M. and Itkonen, J. and Lassenius, C.
- Publication type: Journal/Magazine Article
- Journal: Empirical Software Engineering
- Type of study: Case study (2 cases)

What practices in release engineering does this publication mention?

- Continuous Integration (mainly)
- Code review
- Internal Verification Scope
- Domain Expert Testing
- Testing with customers

Are these practices to be classified under dated, state of the art or state of the practice? Why?

-

What open challenges in release engineering does this publication mention?

- The results in this study can be verified by additional case studies or or even surveys to close the of empirical research on release engineering

Quantitative research publications:

- Study start date:
- Study end date or duration:
- Data acquisition period: 22 weeks (BigCorp) and 24 weeks (SmallCorp)
- Population description:
- Method(s) of recruitment of participants:
- Sample size: 1889 builds (BigCorp) and 760 builds (SmallCorp)

- Evaluation/measurement description:
- Outcomes:
  - High internal quality standards combined with the large distributed organizational context of BigCorp slowed the verification process down and therefore had a negative impact on release capability
  - In SmallCorp, the lack of internal verification measures due to a lack of resources was mitigated by code review, disciplined CI and external verification by customers in customer environments. This allowed for fast release capability and gaining feedback from production.
  - Variables
    - Multiple customers -> High quality standards
    - High quality standards -> Complex CI
    - High quality standards -> Slow Verification
    - Complex CI -> Undisciplined CI
    - Large distributed organization -> Undisciplined CI
    - Undisciplined CI -> Slow verification
    - Slow verification -> Slow release capability
- Limitations:
  - Only a case study, so difficult to generalize
- Future research:

Notes:

- Quantitative results triangulated with interviews

7.5.3.13 Modern Release Engineering in a Nutshell

Reference: [4]

General information:

- Name of person extracting data: Nels Numan
- Date form completed (dd/mm/yyyy): 28/09/2018
- Publication title: Modern Release Engineering in a Nutshell
- Author information: Bram Adams and Shane McIntosh
- Journal: 23rd International Conference on Software Analysis, Evolution, and Reengineering (2016)
- Publication type: Conference paper
- Type of study: Survey

What practices in release engineering does this publication mention?

- Branching and merging
  - Software teams rely on Version Control Systems
  - Quality assurance activities like code reviews are used before doing a merge or even allowing a code change to be committed into a branch
  - Keep branches short-lived and merge often. If this is impossible, a rebase can be done.
  - "trunk-based development" can be applied to eliminate most branches below the master branch.
  - Feature toggles are used to provide isolation for new features in case of the absence of branches.
- Building and testing
  - To help assess build and test conflicts, many projects also provide "try" servers to development teams, which automatically runs a build and test process referred to as CI.
  - The CI process often does not run full test, but a representative subset.
  - The more intensive tests, such as integration, system or performance typically get run nightly or in weekends.
- Build system:
  - GNU Make is the most popular file-based build system technology. Ant is the prototypical task-based build system technology. Lifecycle-based build technologies like Maven consider the build

system of a project to have a sequence of standard build activities that together form a "build lifecycle."
  - "Reproducible builds" involve for a given feature and hardware configuration of the code base, every build invoca- tion should yield bit-to-bit identical build results.
- Infrastructure-as-code
  - Containers or virtual machines are used to deploy new versions of the system for testing or even production.
  - It has been recommended that infrastructure code is to be stored in a separate VCS repository than source code, in order to restrict access to infrastructure code.
- Deployment
  - The term "dark launching" corresponds to deploying new features without releasing them to the public, in which parts of the system automatically make calls to the hidden features in a way invisible to end users.
  - "Blue green deployment" deploys the next software version on a copy of the production environment, and changes this to be the main enviroment on release.
  - In "canary deployment" a prospective release of the software system is loaded onto a subset of the production environments for only a subset of users.
  - "A/B testing" deploys alternative A of a feature to the environment of a subset of the user base, while alternative B is deployed to the environment of another subset.
- Release
  - Once a deployed version of a system is released, the release engineers monitor telemetry data and crash logs to track the performance and quality of releases. Several frameworks and applications have been introduced for this.

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- The majority of these practices are classified by the paper as state of the practice, but state of the art practices are also mentioned.

What open challenges in release engineering does this publication mention?

- Branching and merging
  - No methodology or insight exists on how to empirically validate the best branching structure for a given organization or project, and what results in the smallest amount of merge conflicts.
  - Release engineers need to pay particular attention to conflicts and incompatibilities caused by evolving library and API dependencies.
- Building and testing
  - Speeding up CI might be the major concern of practitioners. This speed up can be achieved through predicting whether a code change will break the build, or by "chunking" code changes into a group and only compile and test each group once.
  - The concept of "green builds" slowly is becoming an issue, in the sense that frequent triggering of the CI server consumes energy.
  - Security of the release engineering pipeline in general, and the CI server in particular, also has become a major concern.
- Release
  - Qualitative studies are not only essential to understand the rationale behind quantitative findings, but also to identify design patterns and best practices for build systems.
    * How can developers make their builds more maintainable and of higher quality?
    * What refactorings should be performed for which build system anti-patterns?
  - Identification and resolution of build bugs, i.e., source code or build specification changes that cause build breakage, possibly on a subset of the supported platforms.
  - Basic tools have a hard time determining what part of the system is necessary to build.
  - Studies on non-GNU Make build systems are missing.
  - Apart from identifying bottlenecks, such approaches should also suggest concrete refactorings of the build system specifications or source code.
- Infrastructure-as-code

  - Research on differences between infrastructure languages is lacking.
  - Best practices and design patterns for infrastructure-as-code need to be documented.
  - Qualitative analysis of infrastructure code will be necessary to understand how developers address different infrastructure needs.
  - Quantitative analysis of the version control and bug report systems can then help to determine which patterns were beneficial in terms of maintenance effort and/or quality.
- Deployment
  - More emperical studies can be done to answer question like this:
    * Is blue-green deployment the fastest means to deploy a new version of a web app?
    * Are A/B testing and dark launching worth the investment and risk?
    * Should one use containers or virtual machines for a medium-sized web app in order to meet application performance and robustness criteria?
    * If an app is part of a suite of apps built around a common database, should each app be deployed in a different container?
  - Better tools for quality assurance are required, to prevent showstopper bugs from slipping through and requiring re-deployment of a mobile app version (with corresponding vetting), these include:
    * Defect prediction (either file- or commit-based)
    * Smarter/safer update mechanisms
    * Tools for improving code review
    * Generating tests
    * Filtering and interpreting crash reports
    * Prioritization and triaging of defect reports
- Release
  - More research is needed on determining which code change is the perfect one for triggering the release of one of these releases, or whether a canary is good enough to be released to another data centre.
  - Question such as the following should be investigated:
    * Should one release on all platforms at the same time?
    * In the case of defects, which platform should receive priority?
    * Should all platforms use the same version numbering, or should that be feature-dependent?
    * Research on the continuous delivery and rapid releases from other systems should be explored.

What research gaps does this publication contain?

- As is common with surveys, it does not contain the state of the field today. More quantitive and qualitive research has been done, which can not possibly be included.

Are these research gaps filled by any other publications in this survey?

- An example of further research that expand on this study is [47]

7.5.3.14 The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues

Reference: [47]

General information:

- Name of person extracting data: Nels Numan
- Date form completed (dd/mm/yyyy): 28/09/2018
- Publication title: The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues
- Author information: Daniel Alencar da Costa, Shane McIntosh, Uira Kulesza, Ahmed E. Hassan
- Journal: 13th Working Conference on Mining Software Repositories (2016)
- Publication type: Conference paper
- Type of study: Emperical study

What practices in release engineering does this publication mention?

- To give a context to the study, the paper describes the concept of traditional releases, rapid releases, their differences, and how issue reports are structured.

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice. The paper describes common practices that were in use at the time of the publication.

What open challenges in release engineering does this publication mention?

- The study mentions that comparing systems with different release structures is difficult since one has to distinguish to what extent the results are due to the release strategy and which are due to intricacies of the systems or organization itself.

What research gaps does this publication contain?

- The main gap in this study is the specificity of the data. Only Mozilla has been considered, and external factors such as other organizational challenges which could have an effect on release time could not be included. More research that looks further into comparing this case to that of other organizations is needed.

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date: Used data starts from 1999
- Study end date or duration: Used data ends in 2010
- Population description: The paper describes multiple steps to describe their data collection approach. The paper collected the date and version number of each Firefox release. Tags within the VCS were used to link issue IDs to releases. The paper discards issues that are potential false positives: IDs that have less five digits, issues that refer to tests instead of bugfixes, any potential ID that is the name of a file. Since the commit logs are linked to the VCS tags, the paper is able to link the issue IDs found within these commit logs to the releases that correspond to those tags.
- Method(s) of recruitment of participants: Firefox release history wiki and VCS logs
- Sample size: 72114 issue reports from the Firefox system (34673 for traditional releases and 37441 for rapid releases)
- Evaluation/measurement description: The paper aims to answer three research questions:
  - Are addressed issues integrated more quickly in rapid releases?
    - Approach: Through beanplots to compare the distributions, the paper first observes the lifetime of the issues of traditional and rapid releases. Next, it looks at the time span of the triaging, fixing, and integration phases within the lifetime of an issue.
  - Why can traditional releases integrate addressed issues more quickly?
    - Approach: the paper groups traditional and rapid releases into major and minor releases and study their integration delay through beanplots, Mann-Whiteney-Wilcoxon tests, Cliff's delta, and MAD.
  - Did the change in the release strategy have an impact on the characteristics of delayed issues?
    - Approach: the paper builds linear regression models for both release approaches. The paper firstly estimates the degrees of freedom that can be spent on the models. Secondly, they check for metrics that are highly correlated using Spearman rank correlation tests and perform a redundancy check to remove redundant metrics. The paper then assesses the fit of our models using the ROC area and the Brier score. The ROC area is used to evaluate the degree of discrimination achieved by the model. The Brier score is used to evaluate the accuracy of probabilistic predictions. The used metrics include reporter experience, resolver experience, issue severity, issue priority, project queue rank, number of impacted files and fix time. A full list of metrics can be found in Table 2 of the paper.

- Outcomes:
  - Are addressed issues integrated more quickly in rapid releases?
    - Results: There is no significant difference between traditional and rapid releases regarding issue lifetime. Results:
  - Why can traditional releases integrate addressed issues more quickly?
    - Results: Minor-traditional releases tend to have less integration delay than major/minor-rapid releases.
  - Did the change in the release strategy have an impact on the characteristics of delayed issues?
    - Results: The models achieve a Brier score of 0.05- 0.16 and ROC areas of 0.81-0.83. Traditional releases prioritize the integration of backlog issues, while rapid releases prioritize the inte- gration of issues of the current release cycle.
- Limitations: Defects in the tools that were developed to perform the data collection and evaluation could have an effect on the outcomes. Furthermore, the way that issue IDs are linked to releases may not represent the total addressed issues per release. The results cannot be generalized as the evaluation was solely done on the Firefox system.
- Future research: Further research can look into applying the same evaluation strategy to other organizations that switched from traditional to rapid release.

Notes:

7.5.3.15 An Empirical Study of Delays in the Integration of Addressed Issues

Reference: [46]

General information:

- Name of person extracting data: Nels Numan
- Date form completed (dd/mm/yyyy): 29/09/18
- Publication title: An Empirical Study of Delays in the Integration of Addressed Issues
- Author information: Daniel Alencar da Costa, Surafel Lemma Abebe, Shane McIntosh, Uira Kulesza, Ahmed E. Hassan
- Journal: 2014 IEEE International Conference on Software Maintenance and Evolution
- Publication type: Conference paper
- Type of study: Emperical study

What practices in release engineering does this publication mention?

- This publication discusses the usage of issue tracking systems, and what the term issue means to form a context around the study.

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- State of the practice.

What open challenges in release engineering does this publication mention?

- The results based on the investigated open source projects may not be generalizable and replication of the study is required on a larger set of projects to form a more general conclusion. Another challenge is finding metrics that are truly correlated with the integration delay of issues.

What research gaps does this publication contain?

- Please see last question.

Are these research gaps filled by any other publications in this survey?

- [47]

Quantitative research publications:

- Study start date:

- Used data start dates:
  - ArgoUML: 18/08/2003
  - Eclipse: 03/11/2003
  - Firefox: 05/06/2012
- Used data end dates:
  - ArgoUML: 15/12/2011
  - Eclipse: 12/02/2007
  - Firefox: 04/02/2014
- Population description:
- Method(s) of recruitment of participants: The data was collected from both ITSs and VCSs of the studied systems.
- Sample size: 20,995 issues from ArgoUML, Eclipse and Firefox projects
- Evaluation/measurement description:
  - How long are addressed issues typically delayed by the integration process?
    - Approach: models are created using metrics from four dimensions: reporter, issue, project, and history. Please refer to Table 2 in the paper for all of the metrics considered. The models are trained using the random forest technique. Precision, recall, F-measure, and ROC area are used to evaluate the models.
- Outcomes:
  - How long are addressed issues typically delayed by the integration process?
    - Addressed issues are usually delayed in a rapid release cycle. Many delayed issues were addressed well before releases from which they were omitted. Many delayed issues were addressed well before releases from which they were omitted.
  - Can we accurately predict when an addressed issue will be integrated?
    - The prediction models achieve a weighted average precision between 0.59 to 0.88 and a recall between 0.62 to 0.88, with ROC areas of above 0.74. The models achieve better F-measure values than Zero-R.
  - What are the most influential attributes for estimating integration delay?
    - The integrator workload has a bigger influence on integrator delay than the other attributes. Severity and priority have little influence on issue in- tegration delay.
- Limitations: See open challenges.
- Future research: See open challenges.

Notes:

7.5.3.16 Towards Definitions for Release Engineering and DevOps

Reference: [61]

General information:

- Name of person extracting data: Nels Numan
- Date form completed (dd/mm/yyyy): 30/09/2018
- Publication title: Towards Definitions for Release Engineering and DevOps
- Author information: Andrej Dyck, Ralf Penners, Horst Lichter
- Journal:
- Publication type:
- Type of study: Survey

What practices in release engineering does this publication mention?

- This paper talks about approaches to improve the collaboration between development and IT operations teams, in order to streamline software engineering processes. The paper defines for release engineering and devops.

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- Not applicable.

What open challenges in release engineering does this publication mention?

- The paper mentions that creating a definition which is uniform and valid for many situations is difficult to find and that further research is needed.

What research gaps does this publication contain?

- This paper aims to form a uniform definition for release engeneering and devops, in collaboration with experts. It is unclear how many experts were consulted for this definition, and more consultations and research could be done to further improve the definition.

Are these research gaps filled by any other publications in this survey?

-

Quantitative research publications:

- Study start date:
- Study end date or duration:
- Population description:
- Method(s) of recruitment of participants:
- Sample size:
- Evaluation/measurement description:
- Outcomes:
- Limitations:
- Future research:

Notes:

7.5.3.17 Continuous deployment of software intensive products and services: A systematic mapping study

Reference: [160]

General information:

- Name of person extracting data: Nels Numan
- Date form completed (dd/mm/yyyy): 30/09/18
- Publication title: Continuous deployment of software intensive products and services: A systematic mapping study
- Author information: Pilar Rodrígueza, Alireza Haghighatkhaha, Lucy Ellen Lwakatarea, Susanna Teppolab, Tanja Suomalainenb, Juho Eskelib, Teemu Karvonena, Pasi Kuvajaa, June M. Vernerc, Markku Oivoa
- Journal:
- Publication type:
- Type of study: Semantic study

What practices in release engineering does this publication mention?

- This paper discussed the developments of continuous development over the years until June 2014. This paper has performed a semantic study to identify, classify and analyze primary studies related to continuous development. The paper has found the following major points:
  - Almost all primary studies make reference in one way or another to accelerate the releae cycle by shortening the release cadence and turning it into a continuous flow.
  - Some reviewed publications claim that accelerating the release cycle can make it harder to perform re-engineering activities.
  - CD challenges and changes traditional planning towards continuous planning in order to achieve fast and frequent releases.

  - Tighter integration between planning and execution is required in order to achieve a more holisitic view on planning in CD.
  - It is important for the engineering and QA teams to ensure backward compatibility of enhancements, so that users perceive only improvements rather than experience any loss of functionality.
  - Code change activities tend to focus more on bug fixing and maintenance than functional- ity expansion
  - The architecture must be robust enough to allow the organization to invest its resources in offensive initiatives such as new functionalitity, product enhancements and innovation rather than defensive efforts such as bugfixes.
  - A major challenge in CD is to retain the balance between speed and quality. Some approaches reviewed by this study propose a focus on measuring and monitoring source code and architectural quality.
  - To avoid issues such as duplicated testing efforts and slow feedback loops it is important to make all testing activities transparent to individual developers.

What open challenges in release engineering does this publication mention?

- Continuous and rapid experimentation is an emerging research topic with many possibilities for future work. This is why it's important to keep up with the newly contributed studies and add them to future reviews to compare their findings.

What research gaps does this publication contain?

-

Notes:

7.5.3.18 Frequent Releases in Open Source Software: A Systematic Review

Reference: [39]

General information:

- Name of person extracting data: Nels Numan
- Date form completed (dd/mm/yyyy): 30/09/18
- Publication title: Frequent Releases in Open Source Software: A Systematic Review
- Author information: Antonio Cesar Brandão Gomes da Silva, Glauco de Figueiredo Carneiro, Fernando Brito e Abreu and Miguel Pessoa Monteiro
- Journal: Information
- Publication type: Journal
- Type of study: Survey

What practices in release engineering does this publication mention?

- This paper discussed the developments of continuous development over the years. This paper has performed a semantic study to identify, classify and analyze primary studies related to continuous development. The paper finds:
  - Two main motivations for the implementation of frequent software releases in the context of OSS projects, which are the project attractiveness/increase of participants and maintenance and increase of market share
  - Four main strategies are adopted by practitioners to implement frequent software releases in the context of OSS projects: time-based release, automated release, test-driven development and continuous delivery/deployment.
  - The main positive points associated to rapid releases are: quick return on customer needs, rapid delivery of new features, quick bug fixes, immediate release security patches, increased efficiency, entry of new collaborators, and greater focus on quality on the part of developers and testers.
  - The main negative points assocaited to rapid releases are reliability of new versions, increase in the "technical debt", pressure felt by employees and community dependence.

Are these practices to be classified under dated, state of the art or state of the practice? Why?

- The practices discussed are a combination of state of the art and state of the practice approaches.

What open challenges in release engineering does this publication mention?

- A meta-model for the mining of open source bases in view of gathering data that leads to assessment of the quality of projects adoping the frequent release approach.

What research gaps does this publication contain?

-

Are these research gaps filled by any other publications in this survey?

-

Chapter 8

Code Review

8.1 Motivation

Code review is a manual assessment process of proposed code changes by other developers than the author, in order to improve code quality and reduce the amount of software defects. As a common software engineering practice, code review is applied in industry and many open-source projects.

Concerning research, code review has become a popular topic recently according to the number of published papers. As noted in [10], the concept of modern code review was proposed in 2013. Since then, many researchers not only explored the modern code review process and its impact on software quality, but also pointed out the possible methods to improve the application of code review.

We collected and read relevant papers, and will present an overview of current research progress in code review in this survey chapter.

8.2 Research protocol

This section describes the review protocol used for the systematic review presented in this chapter. The protocol has been set up using Kitchenham's method as described by [100].

8.2.1 Research questions

The goal of the review is to summarize the state of the art and identify future challenges in the code review area. The research questions are as follows:

- **RQ1**: *What is the state of the art in the research area of code review?* This question focusses on topics that are researched often, the results of that research, and research methods, tools and datasets that are used.
- **RQ2**: *What is the current state of practice in the area of code review?* This concerns tools and techniques that are developed and used in practice, by open source projects but also by commercial companies.
- **RQ3**: *What are future challenges in the area of code review?* This concerns both research challenges and challenges for use in practice.

8.2.2 Search process

The search process consists of the following:

- A Google Scholar search using the search query *"modern code review" OR "modern code reviews"*. The results list will be sorted by decreasing relevance by Google Scholar and will be considered by us in order.
- A general Google search for non-scientific reports (e.g., blog posts) and implemented code review tools. For this search queries *code review* and *code review tools* are used, respectively. The result list will be considered in order.
- All papers in the initial seed provided by the course instructor will be considered.
- All papers referenced by already collected papers will be considered. We exlude papers found using this rule of the search process. In other words, we do not apply this rule recursively.

From now on, elements of all four categories listed above in general will be called *resource*.

8.2.3 Inclusion criteria

From the scientific literature, the following types of papers will be considered:

Papers researching recent code review

- concepts,
- methodologies,
- tools and platforms,
- and experiments concerning the preceding.

From non-scientific resources, all resources discussing recent tools and techniques used in practice will be considered.

8.2.4 Exclusion criteria

Resources published before 2008 will be excluded from the study, in order for the survey to show only the state of the art of the field.

8.2.5 Primary study selection process

We will select a number of candidate resources based on the criteria stated above. For each resource, each person participating in the review can select it as a candidate.

From all candidates, resource will be selected that will actually be reviewed. This can also be done by each person participating in the review. All resources that are candidates but are not selected for actual review must be explicitly rejected, with accompanying reasoning, by at least two persons participating in the review.

8.2.6 Data collection

The following data will be collected from each considered resource:

- Source (for example, the blog website or specific journal)
- Year published
- Type of resource
- Author(s) and organization(s)
- Summary of the resource of a maximum of 100 words
- Data for answering **RQ1**:

  - Sub-topic of research
  - Research method
  - Tools used for research
  - Datasets used for research
  - Research questions and their answers
- Data for answering **RQ2**:
  - Tools used
  - Company/organization using the tool
  - Evaluation of the tool
- Data for answering **RQ3**:
  - Future research challenges posed
  - Challenges for use in practice

All data will be collected by one person participating in the review and checked by another.

8.3 Candidate resources

In the appendix, all candidates that are collected using the described search process are presented. The `In survey` column in the tables indicates whether the paper has been included in the survey in the end or if it has been excluded for some reason. If it has been excluded, the reason will be stated in the section *Excluded papers*.

8.3.1 Initial seed

This table lists all initial seed papers provided by the course instructor that conform to the stated criteria. They are listed in alphabetical order of the first author's name, and then by publish year in Table 1 in the appendix.

8.3.2 Google Scholar

This table lists all candidates that have been collected through the Google Scholar search described in the search process. They are listed in alphabetical order of the first author's name, and then by publish year. Note that as described in the search process section, papers in the search are considered in order of search result number. The *Search date* and *Result number* columns indicate the date on which the search was executed and the position in the search result list, respectively. The table can be found in Table 2 in appendix.

8.3.3 By reference

We list all candidates that have been found by being referenced by another paper we found in Table 3 in appendix.

8.4 Answers

8.4.1 RQ1

RQ1 is *What is the state of the art in the research area of code review?*. As stated in the introduction section, this question concerns itself with topics that are researched often, the results of that research, and

research methods, tools and datasets that are used. Each of these topics will be discussed in the answer to this question.

8.4.1.1 Research methods

First, let us consider the research methods that are generally being used by the papers we incorporated in the survey. The majority of the papers we considered do quantitative research [17, 18, 20, 51, 73, 129, 168, 178, 180, 195] and some qualitative research has also been done [10, 28, 73, 129, 168, 181]. The quantitative research mainly concerns itself with research on open-source projects, while the qualitative research often also considers closed source projects. This is probably the case because the development history, and hence also the code history, is far easier to access for open-source projects than for closed-source projects. The qualitative research mainly concerns itself with interviews, mostly with developers. This is probably the case because it is more convenient to reach developers of proprietary projects, for example because they are often all in one place.

All research that is considered in this survey was done empirically. In other words, no explicit experimental setups have been created just for the purpose of doing the research, but all research has been done on existing situations.

8.4.1.2 Tools and datasets

All papers that do quantitative research use some tools for processing the data. None of the papers in this survey use pre-built tools. All of them have created some custom tools to process the data [17, 18, 20, 51, 73, 168, 178–180, 195]. This mostly concerns custom code in the R programming language that is created for this specific use case only. Some of the papers make the source code of the tools they use available online, so it can be used by other people. The paper by Gousios et al. [73] does this, for example.

As for datasets that are used: there are a few open-source projects that are used very often for quantitative research, notably the following projects: Qt [129, 178–180, 192], Android [178, 180, 181, 192, 195], OpenStack [178–180, 192], LibreOffice [180, 192], Eclipse [192, 195]. Apart from these, mostly other open-source projects have been used, along with a few closed-source projects, for example from Sony Mobile [168] or from Microsoft [195]. What is used from these projects is often only the data from the code reviewing system, possibly along with the code that is under review. As stated above, the reason that open-source projects are used much more as datasets is probably that they are much easier to access.

On another note, the paper by Yang et al. [192] introduces a well structured new dataset for the purpose of performing research and as a benchmark dataset on which code review tools can be tested to compare them. It aims to create a more clear research environment that way.

8.4.1.3 Research subjects

The surveyed papers broadly consider four research subjects, namely factors that the code review process influences, factors that influence the code review process, general characteristics of the code review process, and the performance of tools assisting the code review process. These subjects will be discussed below.

Factors that the code review process influences: Bacchelli and Bird [10] found that code improvement is the most prevalent result of code reviews, followed by code understanding among the development team and social communication within the development team. They note that finding errors is not a prevalent result of doing code reviews, as opposed to what most people participating in it expect from it. In numbers they find that 14% of code review comments were about finding defects, while as much as 44% of the developers inidcates finding defects as the main motivation for doing code reviews.

A bit to the contrary, McIntosh et al. [129] find that low participation in code reviews does lead to a significant increase in post-release defects, which suggests that reviews in which developers show much activity actually

help in finding defects. They additionally find that review coverage, the share of code that has been reviewed, also influences the amount of post-release defects, though not as much as review participation. Shimagaki et al. [168] performed a replication at Sony Mobile study of the just mentioned study by McIntosh et al. Their results were the same for review coverage, and partly for review participation. Contrary to McIntosh et al., they found that the reviewing time and discussion length metrics for review participation did not contribute significantly to the amount of post-release defects.

Thongtanunam et al. [179] back up the claim that doing code reviews helps preventing defects in software, by stating that using code review activity can help to identify defect-prone software modules. They also state that developers who do not often author code changes in the relevant part of code, but still review much can deliver good code reviews. Only when the developer does not author much and does not review much, the code quality can decrease significantly.

Factors that influence the code review process: To start, Baysal et al. [17] found that mainly the experience of the writer of a patch influences the outcome (i.e., accepted or not) of the review. Gousios et al. [73] do not fully agree with this in the context of GitHub pull-requests. They found that only 13% of pull-requests are rejected due to technical reasons, and as much as 53% due to aspects of the distributed nature of pull requests. Thongtanunam et al. [178] add to this that low number of reviewers for prior patches of a patch submitter and a large time since the last modification of the files being modified by the patch, which is also agreed upon by Gousios et al. in the context of pull-requests [73], make it difficult to find reviewers for a patch. Although this does not mean it gets closed immediately, the effect may be the same in the long run. Contrary to what one would expect, they also find that the presence of test code in the patch does not influence the decision to merge it.

Related to this, some submitted patches may simply take a long time to be reviewed. Baysal et al. [17] attribute this to the patch size, which component the patch is for, organizational affiliation of the patch writer, the experience of the patch writer, and the amount of past activity of the reviewer. Bacchelli and Bird [10] note about the last point that understanding the code that is to be reviewed, by the reviewer, is an important challenge. Gousios et al. [73] add to this that the size of the project, its test coverage, and the projects track record on accepting external contributions are also relevant. Thongtanunam et al. [180] add that not being able to find a reviewer for a patch can significantly increase the time required to merge a patch, with an average of 12 days longer. In their research 4%-30% of reviews had this problem, depending on the project. Thongtanunam et al. [178] also note that if a previous path of a submitter took long to review, a new patch is very likely to have the same problem. They also point out that a patch takes longer to merge if the purpose of a patch is to introduce new features. According to Gousios et al. [73], most patches are merged or rejected within one day.

Thongtanunam et al. [178] also found that short patch descriptions, a small code churn, and a small discussion length decrease the chance that a patch will be discussed. Czerwonka, Greiler, and Tilford [51] add to this that when the number of changed files gets above 20, the amount of useful feedback gets lower.

Characteristics of the code review process: Beller et al. [20] found that 75% of the changes in code under review are related to evolvability of the system, and only 25% to its functionality. They also note that the amount of code churn, the number of changed files, and task type determine the number of changes that is done when a patch is under review. According to Gousios et al. [73], most patches are not that big, most being less than 20 lines long (in the context of pull-requests). They also note that discussions are only 3 comments long on average. Beller et al. [20] note about this that 78-90% of the changes that are done during review are because of those comments. The source of the rest of the changes is not known by them.

Another interesting point to note is that only 14% of the repositories on GitHub are actually using pull-requests on GitHub [73]. This may not be readily generalizable to the amount of changes that is being code reviewed in all projects, but is a quite low number nevertheless. Thongtanunam et al. [179] add to this that most developers that only do reviews are core developers, from which one could infer that most patch submissions (and also PRs) would come from external contributors. This together leads one to think that projects are not yet that open to external contributions.

Performance of tools assisting in the code review process: Two tools are proposed in the papers

that have been surveyed: *RevFinder* [180] and *cHRev* [195]. Both tools aim to automatically recommend reviewers to a patch submission, in order to make patch processing faster. RevFinder works by looking at the paths of files that reviewers reviewed previously. It recommends a reviewer whose file path review history looks the most like that of the current patch submission. It uses several string comparison techniques for this. cHRev improves on this by considering how often a reviewer has reviewed changes for a certain component, and also how recently. It has much better accuracy than RevFinder. RevFinder recommends correct reviewers with a median rank of 4 (i.e., a good reviewer candidate is on position 4 on average) based on empirical evaluation. For cHRev, less than 2 recommendations are necessary to find a good reviewer candidate.

8.4.2 RQ2

When it comes to application of code review in industry, we collect information from three perspectives, namely popularity, variety and choices of tools. From collecting information from papers, we know that around one fourth of researched companies regard the code review process as a regular process and about 60 percent of respondents are implementing tool-based code review based on analysis from different companies who are selling code review tools in [14]. Most of the teams use one specialized review tool. One third of the teams choose generic software development tools, like ticket systems and version control systems. Some development teams indicate no tool has been used in their reviews [14]. Considering that there are various tools for code review, we find there are two groups. Specifically, for some teams, no specialized review software is used. Instead, the teams use a combination of IDE, source code management system (SCM) and ticket system/bug tracker. For others, lots of open source tools were used or mentioned: Gerrit, Crucible, Stash, GitHub pull requests, Upsource, Collaborator, ReviewClipse and CodeFlow [15].

Concluding, based on different enterprises' expectation and requirments, they apply various methods for code review. Additionally, we also find different tools are not very comparable as research mentions these are tools for different teams, projects and metrics. It is hard to say which tool is generally better than others. We found that code review is commonly applied in industries and also it is a nice way to guarantee quality of software.

8.4.3 RQ3

What are future challenges in the area of code review? This concerns both research challenges and challenges for use in practice.

Since the concept of modern code review was proposed in 2013 [10], plenty of researchers spend their efforts on exploring code review. According to reference [20, 129], modern code review can be regarded as a lightweight variant of formal code inspections. However, code inspections mandates strict review research criteria and has been proved to improve the software quality. Therefore, in this stage, many papers aim at increasing the understanding of modern code review and figuring out how it improves the software quality. During these study processes, to find out the practical application and impact, qualitative and quantitative methods are applied and some suggested challenges and improvements are found.

- Future research challenges

Firstly, exploration into modern code review is still needed. Many studies suggest that further understanding of modern code review can be helpful to the future research. As an example, in reference [51] it says "Due to its costs, code reviewing practice is a topic deserving to be better understood, systematized and applied to software engineering workflow with more precision than the best practice currently prescribes."

Specifically, some properties of modern code review such as code ownership can be explored, inspired by the reference [129] which proposed a workflow to quantitatively research the relationship between code review coverage and software quality.

In reference [10], awareness and learning during code review are cited as motivations for code review by developers. Future research could research these aspects more explicitly.

Inspired by the progress of the understanding of modern code review, researchers also propose some possible topics that can be explored to obtain more findings.

Bacchelli et al. [10] suggest further research on code comprehension during code review. According to the paper research has been done on this with new developers in mind, but it would also be applicable to code reviews. The authors note that IDEs often include tools for code comprehension, but code review tools do not.

According to reference [51] prior research has neglected the impact of undocumented changes on code review. Future research can focus on this and figure out whether the undocumented changes make a difference.

The authors of reference [73] propose to research on the effect of the democratization of the develoment process, which occurs for example through the use of pull requests. Democratization could for example lead to a substantially stronger commons ecosystem.

They also suggest research on formation of teams and management hierarchies with respect to open-source projects and research on the motives of developers to work in a highly transparent workspace, as prior work do not take these issues into consideration.

Besides, research on studying how best to interpret empirical software engineering research within the context of contextual factors in reference [18]. Understanding the reasons behind observable developer behaviour requires an understanding of the contexts, processes, organizational and individual factors, which can be helpful to realize their influence on code review and the outcome.

- Future challenges in practice

So far, the code review process is adopted both in industry and communities. In reference [10] the authors propose future research on automating code review tasks, which mainly concerns low-level tasks, like checking boundary conditions or catching common mistakes.

Similarly, authors of reference [28] suggest to explore an automatic way to classify and assess the usefulness of comments. This was specifically requested by an interviewees's and is still an open challenge regarding CodeFlow, an in-house code review tool. They also propose to research on methods to automatically recommend reviewers for changes in the system.

In reference [73], the ways to managing tasks in the pull-based development model can be explored, in order to increase the efficiency and readability.

This paper also gives us an example a tool which would suggest whether a pull request can be merged or not, because this can be predicted with fairly high accuracy. Therefore, the development of tools to help the core team of a project with prioritizing their work can be explored.

Several code review tools, such as CodeFlow, ReDA and RevFinder, can still be explored. In reference [180], further research can focus on how RevFinder works in practice, in terms of how effectively and practically it helps developers in recommending code-reviewers, when deployed in a live development environment. According to reference [181], the authors aim to develop a live code review monitoring dashboard based on ReDA. They also aim to create a more portable version of ReDA that is also compatible with other tools supporting the MCR process.

8.5 Conclusions

While answering **RQ1**, we found that doing code reviews can improve social aspects and improve code rubustness against errors. Additionally, finding a reviewer and the experience of a reviewer significantly influence the code review process, along with technical and non-technical aspects of the submitter and patch, like experience or patch description length. On another note, most changes done during code review are

done for maintainability, not for resolving errors, and often the number of comments that are done during review are small. Last, we discussed two tools for recommending reviewers that aim to reduce the time to find appropriate reviewers.

As for **RQ2**, we noticed that tools like Gerrit, Crucible, Stash, GitHub pull requests, Upsource, Collaborator, ReviewClipse and CodeFlow are commonly used in practice. Most reseach proves that code review helps share knowledge and develop defect-free software. It is not very reasonable to rank various tools, as code review itself is much more important for quality of software compared to the choice of a specific tool.

Concerning **RQ3**, we found that as the concept of modern code review was proposed 5 year ago, most of the study aim at increasing the understanding of code review and its impact on software quality. However, prior work were still not enough such that some properties of were not taken into consideration. Therefore, further study on modern code review is still needed, to obtain more useful information. And learned from the research progress, several possible topics based on the prior work that can be explored to obtain more findings. In the practical area, more applications of code review, such as automating code review tasks, are proposed by researchers, which can increase the efficiency. And the code review tools can also be improved according to the current application.

Chapter 9

Runtime and Performance Analytics

In this chapter, we discuss the field of performance and runtime analytics. This chapter however does not cover the entire field because it is too broad. Using Kitchenham's method [102], we have narrowed down the scope of this survey.

For inspiration, we explored five recent papers on runtime and performance analytics published at top conferences. These five were selected because the papers discuss the software side of performance and runtime analytics which is more consistent with the scope of this book. However, focussing on only software, the field is still very broad. We have chosen to focus on the field of performance vs. energy consumption. This choice was made due to it being a very contemporary and thriving domain within runtime and performance analysis.

9.1 Introduction

Energy consumption is an important factor in the day-to-day usage of software. Especially in the field of software development for mobile devices, considering energy consumption determines the battery-life and limites the usage time of the device. Additionally, battery life is seen as very important to most smartphone owners. Ninety two percent of potential smartphone buyers consider battery life as a significant factor in their selection criteria[139]. From the same research it is also found that 66% of smartphone owners would pay more for a device with longer battery life, and 63% is unsatisfied with theirs devices' battery life. Although the importance of energy efficient software is clear, programmers lack the knowledge on the best practices to reduce software energy consumption, and education is not focused on this field either [142]. In an analysis of questions posted on StackOverflow on the topic of energy efficiency performed by Pinto et. al. [147] it was found that although programmers had questions related to energy efficiency, they rarely received appropriate advice. With over 2 billion daily smartphone and tablet users worldwide [56] it is clear that this lack of knowledge needs to be addressed. To help the programmers close this knowledge gap, hardware-based tools have been introduced. These tools can determine the energy profile of an application with high precision, but it means the costly hardware components need to be acquired[56]. Software-based solutions solve the problem of requiring expensive hardware and are easier to use, but also less precise.

In this chapter we represent the current state of energy efficiency in software development for apps. For this we set up the following three research questions:

- **RQ1:** What is the state of the art of energy efficiency in software development for mobile apps?
- **RQ2:** What is the state of practice of energy efficiency in software development for mobile apps?
- **RQ3:** What future work needs to be done in the field of energy efficiency in software development for mobile apps?

In the following paragraph the methodology of the study we have followed will be presented. After that each research question will be answered. In the last paragraph of this chapter the conclusion on energy efficiency in software development for apps will be made.

9.2 Methodology

In order to answer the research questions, we have retrieved over 30 papers from the selected field. These papers have been found by searching on Google Scholar with a set of filters. First off, only papers from the following journals and conferences are selected:

- ACM Transactions on Software Engineering Methodology (TOSEM),
- Empirical Software Engineering (EMSE),
- IEEE Transactions on Software Engineering (TSE),
- Information and Software Technology (IST),
- Journal of Systems and Software (JSS),
- ACM Computing Surveys (CSUR),
- Foundations of Software Engineering (SIGSOFT FSE),
- International Conference on Automated Software Engineering (ASE),
- Working Conference on Mining Software Repositories (MSR)
- Symposium on Operating Systems Design and Implementation (OSDI)

Furthermore, the preference is given to papers which are published not too long ago, preferably after 2012. But if the paper is cited a lot and if it contains interesting findings, older papers can also be used.

Given these two criteria the following search queries have been used:

- "Android AND Energy Efficiency AND Software"
- "Energy AND Android AND Runtime AND Analysis AND performance AND Software"

From the selected papers some backward and forward references have also been selected to get a complete picture. All papers are also checked for relevance to the topic.

To answer **RQ1**, a more in-depth investigation into available tools and guidelines for the energy efficiency is performed. For **RQ2**, the focus lies on what tools, guidelines and best practices are currently being used in mobile software development. **RQ3** is answered by the finding of the papers used for both **RQ1** and **RQ2**.

9.3 RQ1: State of the Art

Many researchers found that there is an increasing demand for energy efficient software. In their research they often propose state of the art approaches which assist developers. These approaches can be separated into two main categories: tools and guidelines. This section will contain an overview of all state of the art tools and guidelines that satisfy our search criteria. This overview is used to answer **RQ1**.

9.3.1 Tools

The table below shows a quick overview of the tools used to answer **RQ1**.

Reference	Tool	Description
[50]	Leafactor	Analyze code smells
[114]	NavyDroid	Locate energy inefficiencies
[97]	Static analysis tool	Identify graphical energy bugs
[162]	APOA	Compare energy consumption of apps

Reference	Tool	Description
[12]	Energy Patch	Detect, validate and repair energy leaks
[55]	PETrA	Measure energy consumption of apps
[144]	jStanley	Detect and improve energy bugs in Java

Leafactor

Using static code analysis and automatic refactoring, Leafactor is able to apply Android-specific optimizations of energy efficiency. The possible optimizations are indicated by a leaf icon, and the fix priority is provided by the official Android lint documentation. This priority reflects the severity of the energy performance, from 1 to 10 with 10 being the most severe energy consumption. Leafactor is able to detect five known energy optimizations.

NavyDroid

NavyDroid is a tool created on top of the Java Pathfinder (JPF). Being constructed as a strengthened DFA (deterministic finite automaton), it can accurately simulate the paused state, the killed state and related state transitions of an activity. This way it can detect complex patterns of wake lock misuses (for example multiple lock aquisitions).

Static analysis tool

The proposed static analysis is a novel static optimization technique for eliminating drawing commands to produce energy-efficient apps. The technique is exploiting the insight that static analysis is able to predict future behavior of the app. With the examples of *loop invariant texture analysis*, *packing* and *identical frames detection* it indicates total energy savings up to 44% of the total energy consumption of the device.

APOA

A recommendation system which can be implemented in any marketplace for helping users and developers to compare apps in terms of performance. As an input APOA uses a set of metrics and rating of apps in Comma Separated Value (CSV) format as well as metrics to optimize (context of usage). The result is a Pareto optimal front, from which the user selects the most preferred solution.

Energy Patch

A framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. It uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. This enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur.

PETrA

A novel software-based tool for measuring the energy consumption of Android apps. PETrA (Power Estimation Tool for Android) measures the energy consumption of Android apps by relying on the tools and APIs provided with the publicly available Project Volta. This means that all smartphones equipped with Android 5.0 or higher are compatible. The tool provides similar performance to hardware-based solutions.

jStanley

Eclipse plugin that helps developers detect energy bugs. jStanley scans the open Java collection for the usage of methods and functions that have more efficient alternatives. Not only are energy leaks highlighted, but better alternatives are provided. The tool is driven by a set of CSV files which contains energy consumption and run time of a wide variety of methods. These values are harvested on a specific device of the researchers. With the help of these values the possible performance boost is calculated.

Most of the state of the art tools can be classified as performing either measurements of energy consumption or code analysis. Although plenty of such tools have been proposed, with low citation numbers, none of them seem to have had a big influence in the research area. There is, however, the possibility, with growing demand for energy efficiency, that the amount and quality of tools will increase in the near future.

9.3.2 Guidelines

Another aspect of the state of the art are the currently proposed guidelines for energy awareness and improvements. To answer **RQ1**, seven papers have been selected that present guidelines. The research by Cruz et al. [49] shows that, interestingly enough, the best practices provided by Google fall short in addressing energy consumption. However they come up with guidelines that do give improvements. Their guideline shows that correct usage of Android methods such as *iewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle* improved energy efficiency. From the research by McIntosh et al.[127] it is noted that machine learning is now also implemented by excited developers into their apps, having an impact on the mobile device's battery life. They have combined empirical measurements of different machine learning algorithm implementations with complexity theory to provide the guideline that some implementations of algorithms, such as J48, MLP and SMO, generally perform better than others. However they also recommend that for optimal results developers must consider their own specific application since many factors, e.g. dataset size, can influence the performance. Finally Li et al.[111] researched commonly recommended energy saving practices, to see if they are valid. The topics investigated in this paper included optimizing the sending of HTTP packets, memory usage, array lengths, static invocations, and field access. This work gives guidance for mobile app developers. Especially bundling network packets up to a certain size and using certain coding practices for reading array length information, accessing class fields, and performing invocations all led to reduced energy consumption.

9.4 RQ2: State of Practice

Now that we have an overview of the state of the art regarding energy consumption on Android driven mobile devices, we investigate what the current state of practice is. In this section, we try to find an answer to **RQ2** stated in the introduction by going through the papers we have retrieved using the methodology described in Section 2. First, we study how different software engineering approaches affect energy consumption and if developers have any knowledge or training regarding energy saving programming techniques. Furthermore, we try to see if there are any tools being used which help decreasing energy consumption. To make this more specific, we divide **RQ2** into several sub-research questions:

- **SQ1:** What are the current development approaches used in practice and how does that affect energy consumption?
- **SQ2:** What do programmers currently know and do about energy saving app development?

9.4.1 Answering SQ1

Under development approaches, the language used to write the app is one of the things that is taken into consideration. There have been a number of studies into the effects of using a certain programming language in the applications. Java is still the most used language for Android applications. According to Oliveira et al. [139], Java is actually not a good choice when limiting energy consumption is one of your priorities. One of the contributions of the paper is a study where hybrid implementations of applications are compared with the original, pure Java implementations in terms of energy consumption and performance. The paper suggests that hybrid implementations might be a good solution, but the problem is that the study was done on just four Android applications. The paper did use the Rosetta Code repository, but this does not represent commercial software development.

Another comparable recent research paper investigated the energy-delay product (EDP) which is defined as "a weighted function of the energy consumption and run-time performance product" [70]:

$E * T^w$

where E is the total energy used to complete a task, T the time and w is a value that represents how important energy saving is with regard to time. The advantage of using this function is that performance is

Rank	Embedded			Laptop	Server
	$w = 1$	$w = 2$	$w = 3$	$w = 1, 2, 3$	$w = 1, 2, 3$
1	C	C	C	C	C
2	C++	C++	C++	Go	Go
3	Go	Go	Go	C++	C++
4	Rust	Rust	Rust	JavaScript	C#
5	C#	C#	JavaScript	Rust	JavaScript
6	VB.NET	JavaScript	C#	C#	Rust
7	JavaScript	VB.NET	VB.NET	VB.NET	VB.NET
8	PHP	PHP	PHP	PHP	PHP
9	Ruby	Ruby	Ruby	Ruby	Python
10	Python	Python	Python	Swift	Ruby
11	Perl	Perl	Perl	Python	Swift
12	Java	Java	Java	Perl	Perl
13	Swift	Swift	Swift	Java	Java
14	R	R	R	R	R

Figure 9.1: Programming Languages Average Weighted EDP Ranking, retrieved from [70]

not neglected. The results in the paper show that the programming language should be chosen depending on the task. From these results, it is clear that Java is not performing well when comparing it to other languages as can be seen in the table below. However, it should be noted that the paper does not test this on actual Android applications but rather the Rosetta Code repository.

The above two papers are examples of research that suggest that the current state of practice, developing applications in Java only, is not in line with the current state of the art on energy efficiency. In spite of these research papers, developers are still working mostly with Java. One possible explanation is that Java is simply compatible with almost every system due to the Java Virtual Machine (JVM). We suggest that further research should investigate why developers are sticking to Java. For example, looking at available libraries, coding difficulty and available tools and knowledge might have a significant impact on this choice.

9.4.2 Answering SQ2

In the previous sections we have seen examples of the state of the art, as well as development methods that could lead to improvements in energy consumption. However, this knowledge only holds value when the community can get developers willing to use these. There have been a number of studies that look into what is actually happening.

One example is the paper by Moura et al. [135]. In this paper a study is conducted by looking at a large number of energy-aware commits with GitHub as the primary data source. This analysis has yielded a list of approaches that are being used by developers in practice. These include: frequency and voltage scaling,

using power efficient libraries and more. The study notes that the vast majority of the commits focus on the lower levels of the software stack. Furthermore, only 16.2% of the commits were related to using more efficient libraries or data structures. There are also a number of software qualities that have been shown to take precedence of the energy consumption. These include correctness, responsiveness and performance. Another point which the research addressed is whether software developers were certain of their energy saving commits. The paper suggests that there are definitely cases where developers are not certain about their energy saving change. This might be caused by the fact that there are few user friendly tools available for aiding developers in making energy-aware decisions. One example of such a tool that attempts to assist the developer in making energy friendly decisions is jStanley [144]. However, both the paper itself and our own replication of that study indicate that this work is not anywhere close to ideal. It only concerns static analysis of the code and is very limited in usability. This prevents such tools of being used in practice. Additionally, there is the problem of how to actually measure the energy being consumed. A number of papers explains how hard this task actually is. Moura et al. [135] also alude to the fact that developers that do use third-party energy-management tools often lack trust in the accuracy of the tools that they are using. The measuring of energy-consumption is not actually as straightforward as one may think. In a number of papers it is explained how hardware-based solutions are accurate, but require expensive components. This is a serious issue, especially in the field of mobile software development. On the other hand, software-based solutions are cheaper but less accurate. Now there is work being done on creating more accurate software solutions [56]. Early work shows promising results in combining the best of both worlds in order to obtain improved results.

9.5 RQ3: Future research

In this section we will take a look at the suggested future research directions that have been mentioned in the papers used in this survey. We attempt to extract some general trends from the future research suggestions that are included. A similarity between almost all of the papers discussed in this section is that they agree the research field is still not very mature. The conclusions of the majority of papers is that more research is required. However, in this survey we found that the consensus is that it is hard to make strong claims, mainly due to having doubts about both the measurement methods as well as the research methodologies.

One of the fields of research we looked at in the previous sections was the relation between programming languages and energy consumption. Multiple papers suggest that the programming language used for implementation has an influence on the energy consumption and performance [70, 139]. These papers however do not look into the influence of specific features of the programming languages on energy consumption and performance. Or rather the specific characteristics of a programming language that cause the differences, as described in the paper by Georgiou et al. [70]

In the investigations of this report, we came across a number of tools which can be used to aid developers in their management of the energy consumption of applications. However, to our knowledge, there are no studies reporting on the actual use of such tools. For future research, we suggest investigating the amount of developers actually using such tools. Furthermore, it is important which features should be included in tools for optimal use. For example, we replicated the paper that introduces jStanley, a tool which can be used for energy and performance optimization [144]. We faced the issue that this tool cannot implement the suggested improvements in an easy and efficient way. For example, we took the AssertJ open source project from GitHub and there were more than 300 energy saving suggestions. A user has to click on every suggestion to actually implement the improvement which is tedious work. Similarly, a lot of the papers reporting on tools that could be used in practice, are currently being tested with benchmarks. However, it is always the question whether these benchmarks are representative enough of the real world to hold actual merit. Additionally, the outcomes of the papers introducing these tools are not comparable. Testing all these tools on the same project would give better insights in what tools are actually useful for the developers to use in terms of gaining as much energy efficiency as possible.

Even though, to our knowledge, there have not been any studies into whether the tools described in the research are actually been used in practice, there have been some studies into energy-saving practices on

GitHub. For instance, studies into the commit messages [13, 135] have shed some light into the state of practice. These studies looked into the commit messages containing keywords related to energy consumption and classified the corresponding code. The future research directions that are indicated by such papers is that it will be important to verify that the energy-saving commits actually have an impact on the overall performance of the software. That is both in terms of energy-consumption as well as metrics for the performance related to usability. Similarly, another direction is verifying that the energy-saving techniques are consistent across platforms.

Finally it is to be noted that nearly all research is based on open-source projects from GitHub repositories. Research on proprietary and closed-source software could possibly lead to different results and would therefore be interesting to conduct.

9.6 Conclusion

Having answered the research questions of this chapter in the sections above, we can conclude that energy awareness with regard to developing applications for mobile devices, and more specific Android, needs more attention. A lot of research is done in the field, resulting in guidelines and tools to help developers. But these guidelines are very generic and also conclude that energy awareness is project specific. The tools presented by papers hold claims to improvements, but for example the tested tool jStanley is not as easy to use as claimed in the paper. More research into the claims made in the papers describing these tools is needed. In practice it is observed that the main language for creating Android apps (Java) is not the most energy efficient option. Furthermore, developers often do not give enough priority to energy saving options and when they do they are often unsure about the effects of their changes.

Chapter 10

App Store Analytics

10.1 Motivation

In the year 2008, the first app stores became available [6][145]. These marketplaces have grown rapidly in size since then with over 3.8 million apps in the Google Play store alone as of first quarter of 2018 [172]. These app stores, together with the large amount of user-generated data associated with them present an unprecedented source of information about the app ecosystem. Software developers and researchers use this valuable data to gain new insights about how users use their app, what they like and the difficulties they encounter. The same data can also be used by the store operators to detect malicious or misbehaving apps whose behavior does not match the description. Because these app stores are relatively new, the research field of App Store Analytics is still not mature. However, because apps are used so much nowadays, it plays a vital role in the field of Software Engineering.

In 2015, Martin, William, et al., published "A survey of app store analysis for software engineering" [121]. The paper divides the field of App Store Analytics into seven categories: API Analysis, Review Analytics, Security, Store Ecosystem, Size and Effort Prediction, Release Engineering, and Feature Analysis. Given that relevant literature on the field of App Store Analytics has been already gathered and analyzed, it makes sense to use this chapter to go a step further. We delve deeper into the subfield of Review Analysis. We started with the literature proposed by Martin et al. in [121] and extended it with the relevant articles that were published after 2015. We then used the data to answer the following research questions:

- **RQ1** What is the state of the art in Review Analysis? Specifically:
  - Which topics that are being explored?
  - Which research methods, tools, and datasets being used?
  - Which are the main research findings (aggregated)?
- **RQ2** What is the state of practice in Review Analysis? Specifically:
  - Which are the tools and the companies that create/employ them?
  - Case studies and their findings.
- **RQ3** Which are the challenges and future research that the field is facing or will face?

10.2 Research Protocol

In this section, we explain the process that we followed to systematically extract the facts from the articles for the literature survey. The search strategy section includes the queries that were used, as well as the initial criteria that ware taken into account for the initial filtering of the articles. Article selection explains how the relevance of the papers was assessed, and how the final filtering was done. The last section describes the data extracted from each article.

10.2.1 Search Strategy

As stated in the motivation, the survey by Martin et al. gave us a starting point. Instead of gathering everything related to App Store Analytics we decided to focus on extending the work done by the authors and answering the research questions specifically for the subcategory of Review Analytics. All the papers that [121] found related with Review Analytics were retrieved and inspected to get a sense of their specific keywords and content.

After doing this, the following search queries were generated:

```
"app store analytics"
"app store analytics" AND "mining"
"app store analytics" AND "user reviews"
"app store analytics" AND "reviews", "app reviews"
```

Google Scholar[1], ACM Digital Library[2], and IEEE Xplore[3] were used for the searching process with the queries mentioned above. In order to build a database using only the relevant articles, the following inclusion criteria were applied to the found articles. In addition to the results obtained by searching with the specific query, the first page of the "related articles" link for the top cited articles was also inspected.

Criteria	Value
time frame	2015-present
journals and conferences	TSE, EMSE, JSS, IST, ICSE, FSE, MSR, OSDI, MobileSoft
keywords in title	user reviews OR app store reviews
keywords in abstract	user reviews OR app store reviews

After the relevant articles were selected, their metadata was gathered, and they were entered into a database. That was the input for the next step of the survey, article selection.

10.2.2 Article selection

Taking into account that the papers considered in this survey were published from 2015, it is no surprise that most of them are not highly cited. As a consequence, the selection of the filtered articles does not take this into consideration. In contrast, each member of the group was in charge of delving into a third of the database and finding, for each study, the *relevance* with respect to Review Analytics and the proposed research questions. Then, this score was peer-reviewed by the other two team members to achieve a consensus.

For the *relevance* score, we considered how much the paper used the analysis of user reviews. Next, we present three examples: 1) a highly relevant paper score=10), 2) a somewhat relevant paper (score=5) and 3) a non-relevant paper (score=0).

What would users change in my app? summarizing app reviews for recommending software changes [57] - Relevance score: 10 - Remarks: the authors applied classification and summarization techniques on app reviews to reduce the effort required to analyze feedback from users. As can be seen, the paper was focused on using the reviews to improve the development process.

[1] https://scholar.google.com/
[2] https://dl.acm.org/
[3] https://ieeexplore.ieee.org/

Fault in your stars: an analysis of Android app reviews [7] - Relevance score: 5 - Remarks: the authors analyzed the problem of the potential mismatch between the app reviews and the star ratings that the app receives. Although it is related to reviews, it is not its primary focus. It also does not push the state of the art regarding the datasets that they used.

Why are Android apps removed from Google Play? A large-scale empirical study [187] - Relevance score: 0 - Remarks: in this case, the paper did not have to do with app reviews even though the title suggested otherwise.

In the end, only the articles that had a score of 5 or more were used for the fact extraction and the subsequent answering of the research questions.

10.2.3 Fact extraction

As was mentioned before, the articles were listed in a database in a structured fashion. The data that was extracted has the following fields:

```
id for indexing
title
year
relevance score
relevance description
source
category
authors Information
source (journal or conference)
complete reference
```

Additionally, for each one of the articles, a systematic reading was carried in which bullet points that answered the following questions were generated:

```
Paper type
Research questions of the paper
Contributions
Datasets: size and sampling methodologies
Techniques used for doing the analysis
```

10.3 Answers

10.3.1 RQ1 What is the state of the art in Review Analysis? Specifically:

- Which topics that are being explored?
- Which research methods, tools, and datasets being used?
- Which are the main research findings (aggregated)?

To answer the question at hand we looked at the novel ideas and the research that has been done in this field since that time. In their survey [121] Martin et al. proposed “Classification”,“Content”,“Requirements Engineering”, “Sentiment”, “Summarization” and “Surveys and Methodological Aspects of App Store Analysis” to categorize the existing literature. After analyzing the subsequent work, we suggest new categories that reflect the state of the art in this field. These are:“Review Manipulation”, “Requirements Engineering”, “Mapping user reviews to the source code”, “Privacy / App Permissions”, “Responding to reviews”, “Comparing Apps and App Stores” and “Wearables”. In the following sections, we expand on each one of these with the revised literature.

Review Manipulation

Recently, significant attention has been paid to how the reviews and ratings can be used to influence the number of downloads of a particular app in the App Store. The paper by Li et al. [112] analyzed the use of crowdsourcing platforms such as Microworkers[4] to manipulate the ratings. The authors merged data from two different sources, App Store and crowdsourcing site, to identify manipulated reviews.

Chen at al. [40] proposes an approach to identify attackers of collusive promotion groups in the app store. They use ranking changes and measurements of pairwise similarity to form targeted app clusters (TAC) that they later use to pinpoint attackers. A different approach to the same problem was proposed in [191]. In the paper, Xie et al., identify manipulated app ratings based on the so-called attack signatures. They present a graph-based algorithm for achieving this purpose in linear time.

These papers also show that the percentage of apps that manipulate their reviews in the app stores is small — less than 1 % of the apps were suspicious [191]. Regarding the datasets, the work by Li et al. [112] used a smaller amount of app store data, but they merged it with data from an external crowdsourcing site. In the case of the other two papers, they can be characterized by the small number of considered apps (compared to the total number of apps in the main marketplaces), but the significant number of reviews that were analyzed. That makes sense, considering that the main purpose is to investigate the later.

Requirements Engineering

Users use reviews as a way to express their attitude (positive and negative) towards an app. The informative value of individual reviews is usually low, but as a whole, they can be mined for knowledge usable in the improvement and advancement of the apps. The survey by Martin et al. [121] described summarization, classification, and requirements engineering as categories of the subfield of Review Analytics. These areas are converging and in recent work (after 2015) summarization and classification, among other techniques, are being used to complement the development, maintenance, and evolution of the apps.

In 2016, Di Sorbo et al. introduced SURF [57], an approach to condense a large number of reviews and extracting useful information from them. In a subsequent paper Panichella et al. [143] presented techniques for classifying useful feedback in the context of maintenance and evolution of the apps. In their work, the authors make use of machine learning supervised methods in conjunction with natural language processing (NLP), sentiment analysis (SA), and text analysis (TA) techniques. Unsupervised methods for review clustering were explored in a paper by Anchiêta et al. [**???**]. The authors introduced a technique to categorize reviews in order to generate a summary of the main bugs and features. Taking into account the high dimensionality of the datasets that are used for review mining, Jha et al. [89] proposed a semantic approach for app review classification. By using semantic role labeling, the authors could make the classification process more efficient. Gao et al. presented IDEA [68]; a framework for identifying emerging app issues based on an online review analysis. IDEA uses information from different time slices (versions) for identification of the issues, and the changelogs of the studied apps as the ground truth for the validation of their approach. It is the only paper in the reviewed literature that presents a case study (of deployment in Tencent) as an example of the viability of IDEA.

Most of the datasets used in the papers analyzed in this section can be characterized by a small number of apps and a substantial amount of reviews. Di Sorbo et al. [57] used 17 apps and 3439 reviews. Anchiêta et al. [**???**] gathered more than 50000 reviews from 3 apps, but after the preprocessing step the dataset was reduced to 924 records. Jha et al. [89] used a structured sampling procedure to mix reviews from different datasets. The final size was 2917 reviews. Finally, Gao et al. [68] used reviews from 6 apps (2 from the App Store and 4 from Google Play), and the final dataset contained 164,026 reviews, one of the largest that had been used.

Mapping user reviews to source code

Another research direction that has become active in the past years is combining the results of mining the app reviews with the source code to directly provide developers with valuable and actionable information to improve their products.

[4] https://www.microworkers.com/

In [141], Palomba et al. presented a new approach called ChangeAdvisor. It extracts useful feedback from reviews and recommends developers changes to specific code artifacts. In the paper, metric distances such as the Dice coefficient are used in order to map a specific subset of reviews to a section of the source code. Complementary work was presented in [140]. In this study, Palomba et al. investigated the extent to which app developers take app reviews into account. To achieve this, they introduced CRISTAL. It pairs informative reviews with source code changes and monitors the extent to which developers accommodate crowd requests from the reviews.

Linting mechanisms and their effectivity have also been studied. Wei et al. [189] proposed a method, OASIS, for prioritizing the Android Lint warnings. It uses NLP techniques (tokenization, word removing, word stemming, and TF-IDF distance) to find the links between the user reviews and the Lint warnings. According to the paper, this is relevant given that one of the problems of linters is the number of false alarms they provide.

Regarding the datasets that were used for validation even though the analyzed papers use a reduced amount of apps (except [140] that used 100), as stated in the previous section, they use numerous reviews (more than 20,000). Additionally, as the aim of the works in this section directly involve the developers, they complement the quantitatively apps-reviews-based experiments with qualitative ones.

Privacy / app permissions

Since Android Marshmallow the Android operating system uses a run-time permission-based security system. Apple's iOS also uses run-time permissions on top of a set of permissions enabled by default. Scoccia et al. did a large-scale empirical study on this new system by inspecting 4.3 million user reviews from 5572 Google Play store apps [165]. Using different techniques they extracted 3574 user reviews that relate to permissions. They found that users like the minimal permissions as most apps only ask for permissions they strictly need. Some of the negative user concerns were apps asking for too many permissions or a bad timing asking for permissions.

Responding to reviews

Since 2013 developers can respond to reviews in the Google Play store and Apple introduced the same feature in 2017. In previous work, McIlroy [125] studied whether responding to user reviews has a positive effect on the rating users give. Building on previous work McIlroy et al. studied how likely it is for users to change their rating for an app when a developer responds to their review [126]. They found that users change their rating 38.7 percent of the time when a developer responds to their review, with a median increase of 20 percent in rating for the app.

Hassan et al. [80] used 2328 top free apps from the Google Play store to study whether users are more likely to update their review if a developer responds to their review. They extracted 126686 dialogues between developers and users and concluded that responding to a review increases the chances of users updating their given rating for an app can increase by up to six times when compared to not responding. They also studied the characteristics, likelihood, and the incentives of user-developer dialogues in app stores.

Comparing Apps and App stores

Papers that compare apps or app stores are discussed in this section. Li et al. mined user reviews from Google Play to find comparisons between apps [113]. They set out to identify comparative reviews to extract differences between apps on different topics. For example, a user says in a review that this app is not as good regarding power consumption than another app. Li et al. created a method that with sufficient accuracy extracts these opinions and provides comparisons between apps.

Ali et al. did a comparative study on cross-platform apps. They took a sample of 80.000 app-pairs to quantitatively compare the same apps across different platforms and identify the differences between the platforms [5].
In a related study Hu et al. compared app-pairs that are created using hybrid development tools such as PhoneGap [85]. With this approach, they found that in 33 of the 68 app-pairs the star rating was not consistent across platforms.

Wearables

User reviews can also be mined to understand user complaints. Mujahid et al. did this in a case study on Android Wear [136]. More specifically, they sampled 589 reviews from 6 Android wearable apps and found that 15 unique complaint types could be extracted from the sample. The sample was created from mining the reviews of 4722 wearable apps and selecting the apps that have more than 100 reviews with a rating of one or two stars. After manually assigning categories to the reviews in the sample they concluded that the most frequent complaints in this relatively small sample were complaints related to cost, functional errors and a lack of functionality.

10.3.2 RQ2 What is the state of practice in Review Analysis? Specifically:

- Which are the tools and the companies that create/employ them?
- Case studies and their findings.

A Large amount of work has been published in the field of app store analytics (The Survey by Martin et al. was done on over 250 papers, and our study analyzed additional 80 papers), only a few of them were case studies. After applying the article selection criteria, we were left with 30 papers from which only two conducted a case study. The first case study of this selection, done by Mujahid et al. [136], examines issues that bother the users of Android Wear. The second one, done by Gao et al. [68], reports on the performance of their review analytics tool which was deployed at Tencent. This creates a large gap between the number of proposed solutions by the researchers and the number of studies done on the solutions that had been deployed in practice.

10.3.2.1 State of the app stores

We were not able to find any academic publication on the solutions used by the actual app stores (Google Play and App Store), but we have found that both app stores automatically detect fake reviews,[5][6]. In 2016 Google deployed their review analytics solution, called Review Highlights[7], into production for both developers and customers, but at the time of writing this survey (October 2018), the Review Highlights no longer show up in the public facing part of Google Play Store and are only accessible in the developer console[8].

10.3.2.2 State of the third-party tools

Many third-party services focus on app store analysis. They usually focus on showing aggregated statistics from the app stores and some of them also analyze the user review. Tools such as AppBot[9] or TheTool[10] perform sentiment analysis on the reviews and show it as an additional metric next to the star rating. AppBot also categorizes reviews based on the keywords inside them. It may seem like a low-tech approach, but it makes it easy for the users to reason about why a certain review is assigned to a specific category. We were not able to find any third-party tools that would use any of the advanced Machine Learning algorithms from the papers we analyzed. This finding combined with Google hiding their Review Highlights may show that most of the algorithms proposed by the researchers do not generalize very well in the real-world application and are hard to comprehend by regular users.

To this day there are, to the best of our knowledge, no tools that compare apps and app stores using user reviews. Some tools analyze apps and app performance, but no tools that do comparisons.

[5] https://www.macrumors.com/2016/10/10/apple-dash-developer-fraudulent-reviews/
[6] https://android-developers.googleblog.com/2016/11/keeping-it-real-improving-reviews-and-ratings-in-google-play.html
[7] https://android-developers.googleblog.com/2016/02/new-tools-for-ratings-reviews-on-google.html
[8] https://support.google.com/googleplay/android-developer/answer/138230?hl=en
[9] https://appbot.co/
[10] https://thetool.io/

10.3.3 RQ3 Which are the challenges and future research that the field is facing or will face?

In this section, we present challenges and future research directions for the field of Review Analytics and the subcategories that were identified in previous sections.

A significant aspect of the research process is the validation of the proposed tools and frameworks and the assessment of how generalizable they are. To accomplish this, it is necessary to have bigger datasets of reviews and more representative and sound samples. Machine learning has seen a steady increase in popularity, however, not so much in the field of Review Analytics. Also, most of the studies rely on correlation relationships to validate the effectiveness of their approaches. There is a need to apply causation techniques so confounding factors can be ruled out.

There is a difficulty in translating the research into actual tools that are used in a real-world setting. Of all the works that were considered, only Gao et al. presented a case study[68], and most of the tools from other papers have either disappeared or are not being actively used.

Review Manipulation: It is important to combine multiple sources of data to identify suspicious individuals better. Not just from app stores alone, but also crowdsourcing sites and even social networks. Also, the sample should be carefully selected, given that the number of suspicious apps is not significant (around 1% of all apps), taking into account the size of the app stores.

Requirements Engineering: More case studies are needed in order to validate how useful the extracted requirements are in the context of software development. Also, as machine learning is starting to be heavily used, models that are tailored to the review data will be created. The large amounts of noise that is present in the reviews is still a challenge that needs careful studying of the preprocessing steps that are used while assembling the datasets.

Mapping user reviews to source code: There is a need to understand the "language of source code" better as this will be essential in combining the review and source code datasets. A likely future trend is the analysis of update-level changes. Regarding this, there is still a need to obtain update-categorized reviews as this is still a challenge with the current review-retrieving approaches.

Privacy/app permissions: Quantifying and understanding the effects of run-time permission requests on the user experience is still an open research area which can help to increase the quality of apps. One of the directions for further research regarding permissions is the idea of giving permissions to specific app functionalities instead of giving permissions to the app as a whole. That could lead to better user-understanding why an app needs certain permission and could reduce the number of permissions an app needs. Another possible future direction is researching and creating tools that help developers put permission requests in the right place to avoid bugs related to permissions and permission requests.

Responding to reviews: There is a need for an in-depth study of how developers and users are using the review mechanism right now to find out how the mechanism can be improved. Right now developers can respond to 500 user reviews per day using the Google Play API[11]. It could be worth investigating whether a limit of 500 responses is sufficient to ban useless responses from the store such as thanking every user.

Comparing Apps and App stores: One of the open challenges is including indirect relationships in the comparisons as only direct relationships were used in work by Li et al. [113]. Next, to this, it has been noted that apps that have been developed using hybrid development tools had relatively low ratings. Future studies should be done to investigate the quality of hybrid apps and compare them with the quality of native apps. Furthermore, to get more complete results, the research should focus on a market-scale analysis using a large number of apps and reviews.

[11] https://developers.google.com/android-publisher/reply-to-reviews

Chapter 11

Final Words

We have finished a nice book on Software Analytics.

.1 Appendix to Chapter 7 (Code Review)

.1.1 Extracted data

This section contains data extracted from all resources included in the survey, according to the *Data collection* section of the review protocol. Note that if some data could not be collected, it is explicitly stated.

The resources are listed in alphabetical order of first author name, and then by year published.

.1.1.1 Expectations, outcomes, and challenges of modern code review

Reference: [10]

Summary

This paper describes research about the goals and actual effects of code reviews. Interviews and experiments have been done with people in the programming field.

One of the main conclusions is that the main effect of doing code reviews is that everyone involved understands the code better. This is opposed to what the goal of code reviews generally is: discovering errors.

For answering **RQ1**:

- *Sub-topic*: in practice; tools
- *Research method*: empirical; qualitative
- *Tools*: N/A
- *Datasets*: Data collected from interviews, surveys and code reviews

Research questions and answers:

- *What are the motivations and expectations for modern code review? Do they change from managers to developers and testers?* The top motivation for code reviews is finding defects, closely followed by code improvement. There does not seem to be a large difference between managers, developers and testers.
- *What are the actual outcomes of modern code review? Do they match the expectations?* Code improvements are the most seen outcomes of code review, followed by code understanding and social

communication. The outcomes do not match the expectations well. For example, only 14% of researched review comments was about code defects, while about 44% chose finding defects as the main motivation for doing code review.
- *What are the main challenges experienced when performing modern code reviews relative to the expectations and outcomes?* The main challenges is by far understanding the code under review. This occurs for example when code has to be reviewed that is not in the same system as a developers works on daily.

For answering **RQ2**:

- *Tools used*: CodeFlow, a reviewing tool. It is not publicly available.
- *Company/organization*: Microsoft
- *Evaluation*: At the time of this paper, it still focusses mainly on fixing errors, and not on the more often ocurring results of doing code review.

For answering **RQ3**:

Future research challenges:

- Research on automating code review tasks. This mainly concerns low-level tasks, like checking boundary conditions or catching common mistakes.
- Research on code comprehension during code review. According to the authors research has been done on this with new developers in mind, but it would also be applicable to code reviews. The authors note that IDEs often include tools for code comprehension, but code review tools do not.
- Research on awareness and learning during code review. Those two aspects were cited as motivations for code review by developers. Future research could research these aspects more explicitly.

.1.1.2 A Faceted Classification Scheme for Change-Based Industrial Code Review Processes

Reference: [15]

Summary
The broad research questions answered in this article are: How is code review performed in industry today? Which commonalities and variations exist between code review processes of different teams and companies? The article describes a classification scheme for change-based code review processes in industry. This scheme is based on descriptions of the code review processes of eleven companies, obtained from interviews with software engineering professionals that were performed during a Grounded Theory study.

.1.1.3 The Choice of Code Review Process: A Survey on the State of the Practice

Reference: [14]

Summary
This paper, published in 2017, is trying to answer 3 RQs. Firstly, how prevalent is change-based review in the industry? Secondly, does the chance that code review remains in use increase if code review is embedded into the process (and its supporting tools) so that it does not require a conscious decision to do a review? Thirdly, are the intended and acceptable levels of review effects a mediator in determining the code review process?

.1.1.4 The influence of non-technical factors on code review

Reference: [18]

Summary
This paper focus on the influence of several non-technical factors on code review response time and outcome. An empirical study of code review process for WebKit, a large open source project was described to see the

influence. Specifically, the authors replicated some previously studied factors and extended several more factors that had not beed explored.

For answering **RQ1**:

- *Sub-topic*: open-source, impact
- *Research method*: empirical study
- *Tools*: WebKit
- *Datasets*: WebKit code review data extracted from Bugzilla.

Research questions and answers:

- *What factors can influence how long it takes for a patch to be reviewed?* The organizational and personal factors influence review timeliness. Some factors that influenced the time required to review a patch, such as the size of the patch itself or the part of the code base being modified, are unsurprising and are likely related to the technical complexity of a given change. The most influential factors of the code review process on review time are the organization a patch writer is affiliated with and their level of participation within the project.
- *What factors influence the outcome of the review process?* The organizational and personal factors influence the likelihood of a patch being accepted. The most influential factors of the code review process on patch acceptance are the organization a patch writer is affiliated with and their level of participation within the project.

For answering **RQ3**:

Future research challenges:

- Research on studying how best to interpret empirical software engineering research within the context of contextual factors. Understanding the reasons behind observable developer behaviour requires an understanding of the contexts, processes, organizational and individual factors that can influence code review and its outcome.

Notes:

This paper has an extended version [17].

.1.1.5 Investigating technical and non-technical factors influencing modern code review

Reference: [17]

Summary:

This article primirarily discusses some non-technical factors that influence the code review process. This are factors like review experience, amount of contributions to a project and company affiliation.

It is found that the most important factors influencing the code review process, in terms of both review time and patch acceptance, are the organization affiliation of the patch writer and the amount of participation of the patch writer in the project.

For answering **RQ1**:

- *Sub-topic*: non-technical
- *Research method*: empirical; quantitative
- *Tools*: Custom
- *Datasets*: WebKit reviews, Google Blink reviews

Research questions and answers:

- *What factors can influence how long it takes for a patch to be reviewed?* "Based on the results of two empirical studies, we found that both technical (patch size and component) , as well as non-technical (organization, patch writer experience, and reviewer activity) factors affect review timeliness when

studying the effect of individual variables. While priority appears to influence review time for WebKit, we were not able to confirm this for Blink."

- *What factors influence the outcome of the review process?* "Our findings from both studies suggest that patch writer experience affects code review outcome. For the WebKit project, factors like priority, organization, and review queue also have an effect on the patch acceptance."

For answering **RQ2**:

- *Tools*: N/A
- *Company/Organization*: N/A
- *Evaluation*: N/A

Notes:

This paper has a shorter version [18].

For answering **RQ3**:

Future research challenges:

Not stated

.1.1.6 Modern code reviews in open-source projects: Which problems do they fix?

Reference: [20]

Summary
It has been researched what kinds of problems are solved by doing code reviews. The conclusion is that 75% are improvements in evolvability of the code, and 25% in functional aspects.

It has also been researched which part of the review comments is actually followed up by an action, and which part of the edits after a review are actually caused by review comments.

For answering **RQ1**:

- *Sub-topic*: impact,changes
- *Research method*: empirically explore; change classification
- *Tools*: R
- *Datasets*: documented history of ConQAT and GROMACS

Research questions and answers:

- *Which types of changes occur in code under review?* 75% of changes are related to the evolvability of the system, and only 25% to its functionality.
- *What triggered the changes occurring in code under review?* 78-90% of the trigger are review comments and the remaining 10-22% are 'undocumented'.
- *What influences the number of changes in code under review?* Code churn, number of changed files and task type are the most important factors influencing the number of changes.

.1.1.7 Lessons learned from building and deploying a code review analytics platform

Reference: [28]

Summary:

A code review data analyzation platform developed and used by Microsoft is discussed. It is mainly presented what users of the system think of it and how its use influences development teams. One of the conclusions is that in general, the platform has a positive influence on development teams and their products.

For answering **RQ2**:

- *Tools used:* CodeFlow, CodeFlow Analytics
- *Company/organization using the tool:* Microsoft
- *Evaluation of the tool:* CodeFlow has already had a positive implace on development teams because of its simplicity, low barrier for feedback and flexible support of Microsoft's disparate engineering systems. But some challenges such as dealing with branches and linking reviews to commits need to improve.

As for CodeFlow Analytics: the tool is being used increasingly throughout Microsoft, with different teams using the tool for different purposes. It is for example effectively used to create dashboards with code review evaluation information, or for examining past reviews in detail. However, some parts of the tool still need to improve in terms of user-friendliness, for example because some functionality is difficult to find.

For answering **RQ3**:

Future research challenges:

- Research on an automatic way to classify and assess the usefulness of comments. This was specifically requested by an interviewees's and is still an open challenge regarding CodeFlow.
- Research on many aspects of code review based on data from CodeFlow Analytics or other similar tools.
- Research on methods to automatically recommend reviewers for changes in the system.

.1.1.8 Software Reviews: The State of the Practice

Reference: [41]

Summary
To investigate how industry carries out software reviews and in what forms, this paper conducted a two-part survey in 2002, the first part based on a national initiative in Germany and the second involving companies worldwide. Additionally, this paper also include some fundamental concepts of code review, such as functionalities of code review.

.1.1.9 Code reviews do not find bugs: how the current code review best practice slows us down

Reference: [51]

Summary
As code review has many uses and benefits, the authors hope to find out whether the current code review methods are sufficiently efficient. They also research whether other methods may be more efficient. With experience gained at Microsoft and with support of data, the authors posit (1) that code reviews often do not find functionality issues that should block a code submission; (2) that effective code reviews should be performed by people with a specific set of skills; and (3) that the social aspect of code reviews cannot be ignored.

For answering **RQ1**:

- *Sub-topic*: impact
- *Research method*: empirical
- *Tools*: not mentioned
- *Datasets*: data collected from engineering systems

Research questions and answers:

- *In what situations, do code reviews provide more value than others?* Unlike inspections, code reviews do not require participants to be in the same place nor do they happen at a fixed, prearranged time. Aligning with a distributed nature of many projects, code reviews are asynchronous and frequently supporting geographically distributed reviewers.

- *What is the value of consistency of applying code reviews equally to all code changes?* Code review usefulness is negatively correlated with the size of a code review. With 20 or more changed files, the more files there are in a single review, the lower the overall rate of useful feedback.

For answering **RQ3**:

Future research challenges:

- Research on undocumented changes of code review because prior research has neglected.
- Due to its costs, code reviewing practice is a topic deserving to be better understood, systematized and applied to software engineering workflow with more precision than the best practice currently prescribes.

.1.1.10 Design and code inspections to reduce errors in program development

Reference: [65]

Summary
This paper describes a method to thoroughly check code quality after each step of the development process, in a heavyweight manner. It does not really concern agile development.

The authors state that these methods do not affect the developing process negatively, and that they work well for improving software quality.

.1.1.11 An exploratory study of the pull-based software development model

Reference: [73]

Summary
This article focuses on how much pull requests are being used and how they are used, focusing on GitHub. For example, it is concluded that pull-requests are not being used that much, that pull-requests are being merged fast after they have been submitted, and that a pull request not being merged is most of the time not caused by technical errors in the pull-request.

For answering **RQ1**:

- *Sub-topic*: open-source, in practice
- *Research method*: empirical; qualitative for finding out reasons for closing pull request, rest quantitative.
- *Tools*: Custom developed tools, available online
- *Datasets*: GHTorrent dataset, along with data collected by authors. The last is also available online

Research questions and answers:

- *How popular is the pull based development model?* "14% of repositories are using pull requests on Github. Pull requests and shared repositories are equally used among projects. Pull request usage is increasing in absolute numbers, even though the proportion of repositories using pull requests has decreased slightly."
- *What are the lifecycle characteristics of pull requests?* "Most pull requests are less than 20 lines long and processed (merged or discarded) in less than 1 day. The discussion spans on average to 3 comments, while code reviews affect the time to merge a pull request. Inclusion of test code does not affect the time or the decision to merge a pull request. Pull requests receive no special treatment, irrespective whether they come from contributors or the core team."
- *What factors affect the decision and the time required to merge a pull request?* "The decision to merge a pull request is mainly influenced by whether the pull request modifies recently modified code. The time to merge is influenced by the developer's previous track record, the size of the project and its test coverage and the project's openness to external contributions."

- *Why are some pull requests not merged?* "53% of pull requests are rejected for reasons having to do with the distributed nature of pull based development. Only 13% of the pull requests are rejected due to technical reasons."

For answering **RQ2**:

- *Tools used*: GitHub PR system
- *Company/organization*: Several open-source projects
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

- More research is needed on *drive-by commits*, which the paper loosely defines as commits added to a repository through a PR by a user that has never contributed to the repository and hence does so for the first time. Often this new contributor also has created a fork for the sole purpose of creating this PR. More research is needed on accurately defining drive-by commits and on assessing their implications.
- More research is needed on the effect of the democratization of the develoment process, which occurs for example through the use of pull requests. Democratization could for example lead to a substantially stronger commons ecosystem.
- Validating the used models on data from different sources and on projects on different languages.
- Research on the motives of developers to work in a highly transparent workspace.
- Research on formation of teams and management hierarchies with respect to open-source projects.
- Research on novel code review practices.
- Research on ways to managing tasks in the pull-based development model.

Challenges in practice:

- Development of tools to help the core team of a project with prioritizing their work. The paper gives as an example a tool which would suggest whether a pull request can be merged or not, because this can be predicted with fairly high accuracy.
- Development of tools that would suggest categories of improvement for pull request, for example by suggesting that more documentation needs to be added.

.1.1.12 The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects

Reference: [129]

Summary
This paper focuses on the influence of doing light-weight code reviews on software quality. In particular, the effect of review coverage (the part of the code that has been reviewed) and review participation (a measure for how much reviewers are involved in the review process) are being assessed.

It turns out that both aspects improve software quality when they are higher. Review participation is the most influential. According to the authors there are other aspects, which they have not looked into, that are of significant importance for the review process.

For answering **RQ1**:

- *Sub-topic*: open-source, in practice, impact
- *Research method*: qualitative for finding out the impact of code review coverage and code review participation on software quality rest quantitative.
- *Tools*: N/A
- *Datasets*: Data extracted from Qt, VTK and ITK code review dataset and necessary metrics including version control metrics, coverage metrics and participation metrics.

Research questions and answers:

- *Is there a relationship between code review coverage and post-release defects?* Although review coverage is negatively associated with software quality in our models, several defect-prone components have high coverage rates, suggesting that other properties of the code review process are at play.
- *Is there a relationship between code review participation and post-release defects?* Lack of participation in code review has a negative impact on software quality. Reviews without discussion are associated with higher post-release defect counts, suggesting that the amount of discussion generated during review should be considered when making integration decisions.

For answering **RQ2**:

- *Tools*: Gerrit
- *Company/Organization*: N/A
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

- Research on other properties of modern code review such as code ownership. Inspired by this paper, other properties of modern code review can also be explored.

Notes:

There exists an extended and improved version of this paper [128]. Only the original version of the paper has been included in this survey.

.1.1.13 A Study of the Quality-Impacting Practices of Modern Code Review at Sony Mobile

Reference: [168]

Summary
First the study by McIntosh et al. [128] is replicated in a proprietary setting at Sony Mobile. A qualitative study, including interviews, is also done with the question "Why are certain reviewing practices associated with better software quality?"

The results from this study are the same as those from the replicated study for RQ1, but not for RQ2. Also, what has been found has been confirmed by the quanitative study has been supported by the qualitative study.

For answering **RQ1**:

- *Sub-topic*:
- *Research method*: replication: empirical, quantitative; qualitative
- *Tools*: N/A
- *Datasets*: Review data from Sony Mobile

Research questions and answers:

- *Is there a relationship between code review coverage and post-release defects?* "Although our review coverage model outperforms our baseline model, of the three studied review coverage metrics, only the proportion of In-House contributions contributes significantly to our model fits. Comparison with previous work. Similar to the prior work [128], we find that Reviewed Commit and Reviewed Churn provide little explanatory power, suggesting that other reviewing factors are at play."
- *Is there a relationship between code review participation and post-release defects?* "Our review participation model also outperforms our baseline model. Of the studied review participation metrics, only the measure of accumulated effort to improve code changes (Patch Sd) and the rate of author self-verification (Self Verify) contribute significantly to our model fits. Comparison with previous work. Unlike the prior work [128], code reviewing time and discussion length did not provide exploratory power to the Sony Mobile model"

For answering **RQ2**:

- *Tools*: Gerrit
- *Company/Organization*: Sony Mobile
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

Not stated

.1.1.14 ReDA: A Web-based Visualization Tool for Analyzing Modern Code Review Dataset

Reference: [181]

Summary:

This paper intoduces *ReDA*, a web-based visualization tool for code review datasets. It processes data from Gerrit, presents statistics about the data, visualizes it, and points the user towards possible problems occurring during the review process. It was tested briefly on some open-source projects.

For answering **RQ1**:

- *Sub-topic*: visualization; tools
- *Research method*: qualitative; empirical
- *Tools*: ReDA
- *Datasets*: Android code review data

Research questions and answers:

N/A

For answering **RQ2**:

- *Tools*: N/A
- *Company/Organization*: N/A
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

The authors aim to develop a live code review monitoring dashboard based on ReDA. They also aim to create a more portable version of ReDA that is also compatible with other tools supporting the MCR process.

.1.1.15 Who should review my code? A file location-based code-reviewer recommendation approach for modern code review

Reference: [180]

Summary:

This paper presents (1) research on how often a reviewer cannot be found for a code change and the influence of this on the time it takes to process a code change, (2) a tool (*RevFinder*) for automatically suggesting reviewers based on files reviewed previously, and (3) an empirical evaluation of that tool on four open-source projects.

Of the researched projects, up to 30% of the code changes have problems finding a reviewer. These reviews take on average 12 days longer. Also, it is found that RevFinder works 3 to 4 times better than an existing tool.

For answering **RQ1**:

- *Sub-topic*: reviewers; tools
- *Research method*: quantitative; empirical
- *Tools*: Custom
- *Datasets*: Custom: Gerrit review data from Android, OpenStack, Qt and LibreOffice

Research questions and answers:

- *How do reviews with code-reviewer assignment problem impact reviewing time?* "4%-30% of reviews have code-reviewer assignment problem. These reviews significantly take 12 days longer to approve a code change. A code-reviewer recommendation tool is necessary in distributed software development to speed up a code review process."
- *Does RevFinder accurately recommend code-reviewers?* "RevFinder correctly recommended 79% of reviews with a top-10 recommendation. RevFinder is 4 times more accurate than ReviewBot. This indicates that leveraging a similarity of previously reviewed file path can accurately recommend code-reviewers."
- *Does RevFinder provide better ranking of recommended code-reviewers?* "RevFinder recommended the correct code-reviewers with a median rank of 4. The code-reviewers ranking of RevFinder is 3 times better than that of ReviewBot, indicating that RevFinder provides a better ranking of correct code-reviewers."

For answering **RQ2**:

- *Tools*: Gerrit
- *Company/Organization*: Google (Android), OpenStack, Qt, The Document Foundation (LibreOffice)
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

Researching how RevFinder works in practice, in terms of how effectively and practically it helps developers in recommending code-reviewers, when deployed in a live development environment.

.1.1.16 Revisiting code ownership and its relationship with software quality in the scope of modern code review

Reference: [179]

Summary:

This paper researches the effect code reviews have on code ownership. This question is answered by looking at two open-source projects. It was found that a lot of contributors do not submit code changes for a specific ticket, but still do quite some reviewing. It was also found that code that contains post-release errors has often been reviewed or authored by people who neither author or review often.

For answering **RQ1**:

- *Sub-topic*: code ownership
- *Research method*: empirical; quantitative
- *Tools*: R; Custom
- *Datasets*: Review dataset from Hamasaki et al. [77]. Code dataset from the Qt system from McIntosh et al. [129]. Ammended with custom datasets for Qt and OpenStack.

Research questions and answers:

- *How do code authoring and reviewing contributions differ?* "The developers who only contribute to a module by reviewing code changes account for the largest set of contributors to that module. Moreover, 18%-50% of these review-only developers are documented core developers of the studied

systems, suggesting that code ownership heuristics that only consider authorship activity are missing the activity of these major contributors."

- *Should code review activity be used to refine traditional code ownership heuristics?* "Many minor authors are major reviewers who actually make large contributions to the evolution of modules by reviewing code changes. Code review activity can be used to refine traditional code ownership heuristics to more accurately identify the defect-prone modules."
- *Is there a relationship between review-specific and review-aware code ownership heuristics and defect-proneness?* "Even when we control for several confounding factors, the proportion of developers in the minor author & minor reviewer category shares a strong relationship with defectproneness. Indeed, modules with a larger proportion of developers without authorship or reviewing expertise are more likely to be defect-prone."

For answering **RQ2**:

- *Tools*: Gerrit
- *Company/Organization*: The Qt, OpenStack, VTK and ITK projects
- *Evaluation*: N/A

.1.1.17 Review participation in modern code review

Reference: [178]

Summary
This paper discusses the factors that influence review participation in code review. Previous studies identified that review participation influences the code review process significantly, but did not study the factors that actually influence review participation.

It was most importantly found that "(…) the review participation history, the description length, the number of days since the last modification of files, the past involvement of an author, and the past involvement of reviewers share a strong relationship with the likelihood that a patch will suffer from poor review participation."

For answering **RQ1**:

- *Sub-topic*: review participation
- *Research method*: empirical; quantitative
- *Tools*: N/A
- *Datasets*: Review data for the Android, Qt and OpenStack projects

Research questions and answers:

- *What patch characteristics share a relationship with the likelihood of a patch not being selected by reviewers?* "We find that the number of reviewers of prior patches, the number of days since the last modification of the patched files share a strong increasing relationship with the likelihood that a patch will have at least one reviewer. The description length is also a strong indicator of a patch that is likely to not be selected by reviewers."
- *What patch characteristics share a relationship with the likelihood of a patch not being discussed?* "We find that the description length, churn, and the discussion length of prior patches share an increasing relationship with the likelihood that a patch will be discussed. We also find that the past involvement of reviewers shares an increasing relationship with the likelihood. On the other hand, the past involvement of an author shares an inverse relationship with the likelihood."
- *What patch characteristics share a relationship with the likelihood of a patch receiving slow initial feedback?* "We find that the feedback delay of prior patches shares a strong relationship with the likelihood that a patch will receive slow initial feedback. Furthermore, a patch is likely to receive slow initial feedback if its purpose is to introduces new features."

For answering **RQ2**:

- *Tools*: Gerrit
- *Company/Organization*: Android, Qt and OpenStack
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

The paper notes that it assumes that the review process is the same for a whole project, even for larger projects. Future work should examine whether there are differences in review processes across subsystems.

.1.1.18 Mining the Modern Code Review Repositories: A Dataset of People, Process and Product

Reference: [192]

Summary:

This paper introduces a dataset that has been systematically collected from review data from several projects. The subject projects are OpenStack, LibreOffice, AOSP, Qt and Eclipse. The dataset is made public for the purpose of doing further research using it. Also, tools may be tested on the data in the dataset, in order to have one benchmark dataset to compare different tools.

For answering **RQ1**:

- *Sub-topic*: tools; dataset
- *Research method*: N/A
- *Tools*: N/A
- *Datasets*: Review data from the OpenStack, LibreOffice, AOSP, Qt and Eclipse projects

Research questions and answers: N/A

For answering **RQ2**:

- *Tools*: Gerrit
- *Company/Organization*: OpenStack, LibreOffice, AOSP, Qt, Eclipse
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

Research using the dataset that has been created, and tests of tools on the dataset.

.1.1.19 Automatically recommending peer reviewers in modern code review

Reference: [195]

Summary:

This paper introduces *cHRev*, a reviewer recommendation approach that, according to the paper, works better in most circumstances than *RevFinder* introduces by Thongtanunam et al. [180]. It recommends reviewers based on their previous review activity. For this it notably uses the frequency of reviews for a specific part of the system and also how recent the reviewing activity was.

For answering **RQ1**:

- *Sub-topic*: reviewer recomendation
- *Research method*: quantitative; empirical

- *Tools*: Custom
- *Datasets*: Reviewing data for Mylyn, Eclipse, Android, and MS Office

Research questions and answers:

- *What is the accuracy of cHRev in recommending reviewers on real software systems across closed and open source projects?* "cHRev makes accurate reviewer recommendations in terms of precision and recall. On average, less than two recommendations are needed to find the first correct reviewer in both closed and open source systems."
- *How do the accuracies of cHRev (trained from the code review history), REVFINDER (also, trained from the code review history, albeit differently), xFinder (trained from the commit history), and RevCom (trained from a combination of the code review and commit histories) compare in recommending code reviewers?* "cHRev performs much better than REVFINDER which is based on reviewers of files with similar names and paths and xFinder which relies on source code repository data, and cHRev is statistically equivalent to RevCom which requires both past reviews and commits."

For answering **RQ2**:

- *Tools*: Gerrit; CodeFlow
- *Company/Organization*: CodeFlow by Microsoft; Gerrit by the other three projects
- *Evaluation*: N/A

For answering **RQ3**:

Future research challenges:

The authors plan to include textual analysis of review comments and additional measures of reviewers' contributions and impact in their approach.

.1.2 Excluded papers

The following papers have been excluded from the survey. These papers are candidates, but have not been added to the final survey for the stated reason.

- [44]: This book is not accessible via the TU Delft subscription of Safari Books Online, and hence we could not read it to include it in the survey.
- [128]: This is an extended and improved version of a paper already included in the survey. Because of time constraints we will not reconsider this version.
- [65]: This paper does not conform to our exclusion criterion saying that it should be published in 2008 or later.

.1.3 Table 1

Title	Year	Reference	In survey? (Y/N)
Expectations, outcomes, and challenges of modern code review	2013	[10]	Y
Modern code reviews in open-source projects: Which problems do they fix?	2014	[20]	Y
Lessons learned from building and deploying a code review analytics platform	2015	[28]	Y
An exploratory study of the pull-based software development model	2014	[73]	Y
The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects	2014	[129]	Y

.1.4 Table 2

Title	Year	Reference	Search date	Result number	In survey? (Y/N)
Investigating technical and non-technical factors influencing modern code review	2016	[17]	29-09-2018	9	Y
Modern code review	2010	[44]	25-09-2018	1	N
An empirical study of the impact of modern code review practices on software quality	2016	[128]	25-09-2018	4	N
A Study of the Quality-Impacting Practices of Modern Code Review at Sony Mobile	2016	[168]	29-09-2018	11	Y
Reda: A web-based visualization tool for analyzing modern code review dataset	2014	[181]	29-09-2018	8	Y
Who should review my code? A file location-based code-reviewer recommendation approach for modern code review	2015	[180]	29-09-2018	5	Y
Revisiting code ownership and its relationship with software quality in the scope of modern code review	2016	[179]	29-09-2018	6	Y
Review participation in modern code review	2017	[178]	29-09-2018	10	Y
Mining the Modern Code Review Repositories: A Dataset of People, Process and Product	2016	[192]	29-09-2018	12	Y
Automatically recommending peer reviewers in modern code review	2016	[195]	29-09-2018	7	Y

.1.5 Table 3

Title	Year	Reference	In survey? (Y/N)
A Faceted Classification Scheme for Change-Based Industrial Code Review Processes	2016	[15]	Y
The Choice of Code Review Process: A Survey on the State of the Practice	2017	[14]	Y
The influence of non-technical factors on code review	2013	[18]	Y
Impact of peer code review on peer impression formation: A survey	2013	[32]	N
Software Reviews: The State of the Practice	2003	[41]	N
Code reviews do not find bugs: how the current code review best practice slows us down	2015	[51]	Y

[1] Abate, P. and Cosmo, R.D. 2011. Predicting upgrade failures using dependency analysis. *2011 IEEE 27th international conference on data engineering workshops* (Apr. 2011).

[2] Abate, P. et al. 2009. Strong dependencies between software components. *2009 3rd international symposium on empirical software engineering and measurement* (Oct. 2009).

[3] Abdalkareem, R. et al. 2017. Why do developers use trivial packages? An empirical case study on npm.

Proceedings of the 2017 11th joint meeting on foundations of software engineering - ESEC/FSE 2017 (2017).

[4] Adams, B. and McIntosh, S. 2016. Modern release engineering in a nutshell–why researchers should care. *Software analysis, evolution, and reengineering (saner), 2016 ieee 23rd international conference on* (2016), 78–90.

[5] Ali, M. et al. 2017. Same app, different app stores: A comparative study. *Proceedings of the 4th international conference on mobile software engineering and systems* (2017), 79–90.

[6] AppleInsider 2008. Apple's app store launches with more than 500 apps. http://appleinsider.com/articles/08/07/10/apples_app_store_launches_with_more_than_500_apps.

[7] Aralikatte, R. et al. 2018. Fault in your stars: An analysis of android app reviews. *Proceedings of the acm india joint international conference on data science and management of data* (2018), 57–66.

[8] Arisholm, E. et al. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*. 83, 1 (2010), 2–17.

[9] Atifi, M. et al. 2017. *A comparative study of software testing techniques*.

[10] Bacchelli, A. and Bird, C. 2013. Expectations, outcomes, and challenges of modern code review. *Proceedings of the 2013 international conference on software engineering* (2013), 712–721.

[11] Baltes, S. et al. 2018. (No) influence of continuous integration on the commit activity in github projects. *arXiv preprint arXiv:1802.08441*. (2018).

[12] Banerjee, A. et al. 2018. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*. 44, 5 (2018), 470–490.

[13] Bao, L. et al. 2016. How android app developers manage power consumption?: An empirical study by mining power management commits. *Proceedings of the 13th international conference on mining software repositories* (2016), 37–48.

[14] Baum, T. et al. 2017. The choice of code review process: A survey on the state of the practice. *International conference on product-focused software process improvement* (2017), 111–127.

[15] Baum, T. et al. 2016. A faceted classification scheme for change-based industrial code review processes. *Software quality, reliability and security (qrs), 2016 ieee international conference on* (2016), 74–85.

[16] Bavota, G. et al. 2014. How the apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering*. 20, 5 (Sep. 2014), 1275–1317.

[17] Baysal, O. et al. 2016. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*. 21, 3 (2016), 932–959.

[18] Baysal, O. et al. 2013. The influence of non-technical factors on code review. *Reverse engineering (wcre), 2013 20th working conference on* (2013), 122–131.

[19] Beck, K. 2003. *Test-driven development: By example*. Addison-Wesley Professional.

[20] Beller, M. et al. 2014. Modern code reviews in open-source projects: Which problems do they fix? *Proceedings of the 11th working conference on mining software repositories* (2014), 202–211.

[21] Beller, M. et al. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*. 1 (2017), 1–1.

[22] Beller, M. et al. 2015. How (much) do developers test? *Proceedings of the 37th international conference on software engineering - volume 2* (Piscataway, NJ, USA, 2015), 559–562.

[23] Beller, M. et al. 2017. Oops, my tests broke the build: An explorative analysis of travis ci with github. *Mining software repositories (msr), 2017 ieee/acm 14th international conference on* (2017), 356–367.

[24] Beller, M. et al. 2017. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. *Proceedings of the 14th international conference on mining software repositories*

(2017), 447–450.

[25] Beller, M. et al. 2015. When, how, and why developers (do not) test in their ides. *2015 10th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering, esec/fse 2015 - proceedings* (2015), 179–190.

[26] Bevan, J. et al. 2005. Facilitating software evolution research with kenyon. *ESEC/fse'05 - proceedings of the joint 10th european software engineering conference (esec) and 13th acm sigsoft symposium on the foundations of software engineering (fse-13)* (2005), 177–186.

[27] Bird, C. and Zimmermann, T. 2017. Predicting software build errors. Google Patents.

[28] Bird, C. et al. 2015. Lessons learned from building and deploying a code review analytics platform. *Proceedings of the 12th working conference on mining software repositories* (2015), 191–201.

[29] Bisong, E. et al. 2017. Built to last or built too fast?: Evaluating prediction models for build times. *Proceedings of the 14th international conference on mining software repositories* (2017), 487–490.

[30] Blincoe, K. et al. 2015. Ecosystems in GitHub and a method for ecosystem identification using reference coupling. *2015 IEEE/ACM 12th working conference on mining software repositories* (May 2015).

[31] Bogart, C. et al. 2016. How to break an API: Cost negotiation and community values in three software ecosystems. *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering - FSE 2016* (2016).

[32] Bosu, A. and Carver, J.C. 2013. Impact of peer code review on peer impression formation: A survey. *Empirical software engineering and measurement, 2013 acm/ieee international symposium on* (2013), 133–142.

[33] Bouwers, E. et al. 2012. Getting what you measure. *Commun. ACM.* 55, 7 (Jul. 2012), 54–59.

[34] Bowring, J. and Hegler, H. 2014. Obsidian: Pattern-based unit test implementations. *Journal of Software Engineering and Applications.* 7, 02 (2014), 94.

[35] Castelluccio, M. et al. 2017. Is it safe to uplift this patch? An empirical study on mozilla firefox. *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017* (2017), 411–421.

[36] Catal, C. 2011. Software fault prediction: A literature review and current trends. *Expert Systems with Applications.* 38, 4 (2011), 4626–4636.

[37] Catal, C. and Diri, B. 2009. A systematic review of software fault prediction studies.

[38] Catal, C. and Diri, B. 2009. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences.* 179, 8 (2009), 1040–1058.

[39] Cesar Brandão Gomes da Silva, A. et al. 2017. Frequent releases in open source software: A systematic review. *Information.* 8, 3 (2017), 109.

[40] Chen, H. et al. 2017. Toward detecting collusive ranking manipulation attackers in mobile app markets. *Proceedings of the 2017 acm on asia conference on computer and communications security* (2017), 58–70.

[41] Ciolkowski, M. et al. 2003. Software reviews: The state of the practice. *IEEE software.* 6 (2003), 46–51.

[42] Claes, M. et al. 2017. Abnormal working hours: Effect of rapid releases and implications to work content. *IEEE International Working Conference on Mining Software Repositories* (2017), 243–247.

[43] Claes, M. et al. 2015. A historical analysis of debian package incompatibilities. *2015 IEEE/ACM 12th working conference on mining software repositories* (May 2015).

[44] Cohen, J. 2010. Modern code review. *Making Software: What Really Works, and Why We Believe It.* (2010), 329–336.

[45] Constantinou, E. and Mens, T. 2017. An empirical comparison of developer retention in the RubyGems

and npm software ecosystems. *Innovations in Systems and Software Engineering.* 13, 2-3 (Aug. 2017), 101–115.

[46] Costa, D.A. da et al. 2014. An empirical study of delays in the integration of addressed issues. *2014 ieee international conference on software maintenance and evolution* (2014), 281–290.

[47] Costa, D.A. da et al. 2016. The impact of switching to a rapid release cycle on the integration delay of addressed issues - an empirical study of the mozilla firefox project. *2016 ieee/acm 13th working conference on mining software repositories (msr)* (2016), 374–385.

[48] Cox, J. et al. 2015. Measuring dependency freshness in software systems. *2015 IEEE/ACM 37th IEEE international conference on software engineering* (May 2015).

[49] Cruz, L. and Abreu, R. 2017. Performance-based guidelines for energy efficient mobile applications. *Mobile software engineering and systems (mobilesoft), 2017 ieee/acm 4th international conference on* (2017), 46–57.

[50] Cruz, L. and Abreu, R. 2018. Using automatic refactoring to improve energy efficiency of android apps. *arXiv preprint arXiv:1803.05889.* (2018).

[51] Czerwonka, J. et al. 2015. Code reviews do not find bugs: How the current code review best practice slows us down. *Proceedings of the 37th international conference on software engineering-volume 2* (2015), 27–28.

[52] Decan, A. et al. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)* (Feb. 2017).

[53] Decan, A. et al. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering.* (Feb. 2018).

[54] Di Nucci, D. et al. 2018. A developer centered bug prediction model. *IEEE Transactions on Software Engineering.* 44, 1 (2018), 5–24.

[55] Di Nucci, D. et al. 2017. Petra: A software-based tool for estimating the energy profile of android applications. *Proceedings of the 39th international conference on software engineering companion* (2017), 3–6.

[56] Di Nucci, D. et al. 2017. Software-based energy profiling of android apps: Simple, efficient and reliable? *Software analysis, evolution and reengineering (saner), 2017 ieee 24th international conference on* (2017), 103–114.

[57] Di Sorbo, A. et al. 2016. What would users change in my app? Summarizing app reviews for recommending software changes. *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (2016), 499–510.

[58] Dietrich, J. et al. 2014. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. *2014 software evolution week - IEEE conference on software maintenance, reengineering, and reverse engineering (CSMR-WCRE)* (Feb. 2014).

[59] Dittrich, Y. 2014. Software engineering beyond the project sustaining software ecosystems. *Information and Software Technology.* 56, 11 (Nov. 2014), 1436–1456.

[60] Dulz, W. 2013. Model-based strategies for reducing the complexity of statistically generated test suites. *International conference on software quality* (2013), 89–103.

[61] Dyck, A. et al. 2015. Towards definitions for release engineering and devops. *Release engineering (releng), 2015 ieee/acm 3rd international workshop on* (2015), 3–3.

[62] D'Ambros, M. et al. 2010. An extensive comparison of bug prediction approaches. *Proceedings - International Conference on Software Engineering.* (2010), 31–41.

[63] D'Ambros, M. et al. 2012. Evaluating defect prediction approaches: A benchmark and an extensive

comparison. *Empirical Software Engineering*. 17, 4-5 (2012), 531–577.

[64] Eick, S.G. et al. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*. 27, 1 (Jan. 2001), 1–12.

[65] Fagan, M. 2002. Design and code inspections to reduce errors in program development. *Software pioneers*. Springer. 575–607.

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

[67] Fujibayashi, D. et al. 2017. Does the release cycle of a library project influence when it is adopted by a client project? *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (2017), 569–570.

[68] Gao, C. et al. 2018. Online app review analysis for identifying emerging issues. *2018 ieee/acm 40th international conference on software engineering (icse)* (2018), 48–58.

[69] Garousi, V. and Zhi, J. 2013. A survey of software testing practices in canada. *Journal of Systems and Software*. 86, 5 (2013), 1354–1376.

[70] Georgiou, S. et al. 2018. What are your programming language's energy-delay implications? *Proceedings of the 15th international conference on mining software repositories* (2018), 303–313.

[71] Giger, E. et al. 2012. Method-level bug prediction. *Proceedings of the acm-ieee international symposium on empirical software engineering and measurement* (New York, NY, USA, 2012), 171–180.

[72] Giger, E. et al. 2011. Comparing fine-grained source code changes and code churn for bug prediction. *Proceedings of the 8th working conference on mining software repositories* (New York, NY, USA, 2011), 83–92.

[73] Gousios, G. et al. 2014. An exploratory study of the pull-based software development model. *Proceedings of the 36th international conference on software engineering* (2014), 345–355.

[74] Greiler, M. et al. 2013. Strategies for avoiding text fixture smells during software evolution. *IEEE international working conference on mining software repositories* (2013), 387–396.

[75] Gyimothy, T. et al. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*. 31, 10 (Oct. 2005), 897–910.

[76] Hall, T. et al. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*. 38, 6 (Nov. 2012), 1276–1304.

[77] Hamasaki, K. et al. 2013. Who does what during a code review? Datasets of oss peer review repositories. *Proceedings of the 10th working conference on mining software repositories* (2013), 49–52.

[78] Hassan, A.E. 2009. Predicting faults using the complexity of code changes. *Proceedings of the 31st international conference on software engineering* (Washington, DC, USA, 2009), 78–88.

[79] Hassan, F. and Wang, X. 2018. HireBuild: An automatic approach to history-driven repair of build scripts. *Proceedings of the 40th international conference on software engineering* (2018), 1078–1089.

[80] Hassan, S. et al. 2018. Studying the dialogue between users and developers of free apps in the google play store. *Empirical Software Engineering*. 23, 3 (2018), 1275–1312.

[81] Hejderup, J. et al. 2018. Software ecosystem call graph for dependency management. *Proceedings of the 40th international conference on software engineering new ideas and emerging results - ICSE-NIER 18* (2018).

[82] Hemmati, H. and Sharifi, F. 2018. Investigating nlp-based approaches for predicting manual test case failure. *Proceedings - 2018 ieee 11th international conference on software testing, verification and validation, icst 2018* (2018), 309–319.

[83] Hilton, M. et al. 2016. Usage, costs, and benefits of continuous integration in open-source projects.

Proceedings of the 31st ieee/acm international conference on automated software engineering (2016), 426–437.

[84] Hora, A. et al. 2016. How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*. 26, 1 (Oct. 2016), 161–191.

[85] Hu, H. et al. 2018. Studying the consistency of star ratings and reviews of popular free hybrid android and iOS apps. *Empirical Software Engineering*. (2018), 1–26.

[86] Hurdugaci, V. and Zaidman, A. 2012. Aiding software developers to maintain developer tests. *2012 16th european conference on software maintenance and reengineering* (March 2012), 11–20.

[87] Izquierdo, D. et al. 2018. Software development analytics for xen: Why and how. *IEEE Software*. (2018), 1–1.

[88] Jansen, S. 2014. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*. 56, 11 (Nov. 2014), 1508–1519.

[89] Jha, N. and Mahmoud, A. 2017. Mining user requirements from application store reviews using frame semantics. *International working conference on requirements engineering: Foundation for software quality* (2017), 273–287.

[90] Jiang, Y. et al. 2008. Techniques for evaluating fault prediction models. *Empirical Software Engineering*. 13, 5 (Oct. 2008), 561–595.

[91] Karvonen, T. et al. 2017. Systematic literature review on the impacts of agile release engineering practices. *Information and Software Technology*. 86, (2017), 87–100.

[92] Kaur, A. and Vig, V. 2019. On understanding the release patterns of open source java projects. *Advances in Intelligent Systems and Computing*. 711, (2019), 9–18.

[93] Kerzazi, N. and Robillard, P. 2013. Kanbanize the release engineering process. *2013 1st International Workshop on Release Engineering, RELENG 2013 - Proceedings* (2013), 9–12.

[94] Khomh, F. et al. 2015. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*. 20, 2 (2015), 336–373.

[95] Khomh, F. et al. 2012. Do faster releases improve software quality?: An empirical case study of mozilla firefox. *Proceedings of the 9th ieee working conference on mining software repositories* (Piscataway, NJ, USA, 2012), 179–188.

[96] Kikas, R. et al. 2017. Structure and evolution of package dependency networks. *2017 IEEE/ACM 14th international conference on mining software repositories (MSR)* (May 2017).

[97] Kim, C.H.P. et al. 2016. Static program analysis for identifying energy bugs in graphics-intensive mobile apps. *Modeling, analysis and simulation of computer and telecommunication systems (mascots), 2016 ieee 24th international symposium on* (2016), 115–124.

[98] Kim, S. et al. 2011. Dealing with noise in defect prediction. *Proceedings of the 33rd international conference on software engineering* (New York, NY, USA, 2011), 481–490.

[99] Kim, S. et al. 2007. Predicting faults from cached history. *Proceedings of the 29th international conference on software engineering* (Washington, DC, USA, 2007), 489–498.

[100] Kitchenham 2007. *Guidelines for performing systematic literature reviews in software engineering*. Keele University; University of Durham.

[101] Kitchenham, B. 2004. Procedures for performing systematic reviews. *Keele*. 33, 1 (2004), 1–26.

[102] Kitchenham, B. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University*. 33, 2004 (2004), 1–26.

[103] Kula, R.G. et al. 2017. An exploratory study on library aging by monitoring client usage in a software ecosystem. *2017 IEEE 24th international conference on software analysis, evolution and reengineering*

(SANER) (Feb. 2017).

[104] Kula, R.G. et al. 2017. Do developers update their library dependencies? *Empirical Software Engineering*. 23, 1 (May 2017), 384–417.

[105] Laukkanen, E. et al. 2017. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*. 82, (2017), 55–79.

[106] Laukkanen, E. et al. 2018. Comparison of release engineering practices in a large mature company and a startup. *Empirical Software Engineering*. (2018), 1–43.

[107] Lee, T. et al. 2011. Micro interaction metrics for defect prediction. *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (New York, NY, USA, 2011), 311–321.

[108] Lessmann, S. et al. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*. 34, 4 (2008), 485–496.

[109] Leung, H.K. and Lui, K.M. 2015. Testing analytics on software variability. *Software analytics (swan), 2015 ieee 1st international workshop on* (2015), 17–20.

[110] Lewis, C. et al. 2013. Does bug prediction support human developers? Findings from a Google case study. *2013 35th international conference on software engineering (icse)* (May 2013), 372–381.

[111] Li, D. and Halfond, W.G. 2014. An investigation into energy-saving programming practices for android smartphone app development. *Proceedings of the 3rd international workshop on green and sustainable software* (2014), 46–53.

[112] Li, S. et al. 2017. Crowdsourced app review manipulation. *Proceedings of the 40th international acm sigir conference on research and development in information retrieval* (2017), 1137–1140.

[113] Li, Y. et al. 2017. Mining user reviews for mobile app comparisons. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*. 1, 3 (2017), 75.

[114] Liu, Y. et al. 2017. NavyDroid: Detecting energy inefficiency problems for smartphone applications. *Proceedings of the 9th asia-pacific symposium on internetware* (2017), 8.

[115] Lungu, M. 2009. *Reverse engineering software ecosystems*. University of Lugano.

[116] Malhotra, R. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*. 27, C (Feb. 2015), 504–518.

[117] Malloy, B.A. and Power, J.F. 2018. An empirical analysis of the transition from python 2 to python 3. *Empirical Software Engineering*. (Jul. 2018).

[118] Malloy, B.A. and Power, J.F. 2017. Quantifying the transition from python 2 to 3: An empirical study of python applications. *2017 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)* (Nov. 2017).

[119] Manikas, K. 2016. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*. 117, (Jul. 2016), 84–103.

[120] Marsavina, C. et al. 2014. Studying fine-grained co-evolution patterns of production and test code. *2014 ieee 14th international working conference on source code analysis and manipulation* (Sept 2014), 195–204.

[121] Martin, W. et al. 2017. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*. 43, 9 (2017), 817–847.

[122] Matsumoto, S. et al. 2010. An analysis of developer metrics for fault prediction. *Proceedings of the 6th international conference on predictive models in software engineering* (New York, NY, USA, 2010), 18:1–18:9.

[123] Mäntylä, M.V. et al. 2015. On rapid releases and software testing: A case study and a semi-systematic

literature review. *Empirical Software Engineering*. 20, 5 (2015), 1384–1425.

[124] McDonnell, T. et al. 2013. An empirical study of API stability and adoption in the android ecosystem. *2013 IEEE international conference on software maintenance* (Sep. 2013).

[125] Mcilroy, S. 2014. *Empirical studies of the distribution and feedback mechanisms of mobile app stores*.

[126] McIlroy, S. et al. 2017. Is it worth responding to reviews? Studying the top free apps in google play. *IEEE Software*. 34, 3 (2017), 64–71.

[127] McIntosh, A. et al. 2018. What can android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering*. (2018), 1–40.

[128] McIntosh, S. et al. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*. 21, 5 (2016), 2146–2189.

[129] McIntosh, S. et al. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. *Proceedings of the 11th working conference on mining software repositories* (2014), 192–201.

[130] Mens, T. et al. 2013. Studying evolving software ecosystems based on ecological models. *Evolving software systems*. Springer Berlin Heidelberg. 297–326.

[131] Messerschmitt, D.G. and Szyperski, C. 2003. *Software ecosystem: Understanding an indispensable technology and industry (mit press)*. The MIT Press.

[132] Mirzaaghaei, M. et al. 2012. Supporting test suite evolution through test case adaptation. *2012 ieee fifth international conference on software testing, verification and validation* (April 2012), 231–240.

[133] Moiz, S.A. 2017. Uncertainty in software testing. *Trends in software testing*. Springer. 67–87.

[134] Moser, R. et al. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *Proceedings of the 30th international conference on software engineering* (New York, NY, USA, 2008), 181–190.

[135] Moura, I. et al. 2015. Mining energy-aware commits. *Proceedings of the 12th working conference on mining software repositories* (2015), 56–67.

[136] Mujahid, S. et al. 2017. Examining user complaints of wearable apps: A case study on android wear. *Mobile software engineering and systems (mobilesoft), 2017 ieee/acm 4th international conference on* (2017), 96–99.

[137] Ni, A. and Li, M. 2018. ACONA: Active online model adaptation for predicting continuous integration build failures. *Proceedings of the 40th international conference on software engineering: Companion proceeedings* (2018), 366–367.

[138] Noor, T.B. and Hemmati, H. 2015. Test case analytics: Mining test case traces to improve risk-driven testing. *Software analytics (swan), 2015 ieee 1st international workshop on* (2015), 13–16.

[139] Oliveira, W. et al. 2017. A study on the energy consumption of android app development approaches. *Mining software repositories (msr), 2017 ieee/acm 14th international conference on* (2017), 42–52.

[140] Palomba, F. et al. 2018. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software*. 137, (2018), 143–162.

[141] Palomba, F. et al. 2017. Recommending and localizing change requests for mobile apps based on user reviews. *Proceedings of the 39th international conference on software engineering* (2017), 106–117.

[142] Pang, C. et al. 2016. What do programmers know about software energy consumption? *IEEE Software*. 33, 3 (2016), 83–89.

[143] Panichella, S. et al. 2016. Ardoc: App reviews development oriented classifier. *Proceedings of the 2016*

24th acm sigsoft international symposium on foundations of software engineering (2016), 1023–1027.

[144] Pereira, R. et al. 2018. JStanley: Placing a green thumb on java collections. *Proceedings of the 33rd acm/ieee international conference on automated software engineering* (2018), 856–859.

[145] Perenson, M. 2008. Google launches android market. https://www.pcworld.com/article/152613/google_android_ships.html.

[146] Pinto, G. and Rebouças, F.C.R.B.M. 2018. Work practices and challenges in continuous integration: A survey with travis ci users. (2018).

[147] Pinto, G. et al. 2014. Mining questions about software energy consumption. *Proceedings of the 11th working conference on mining software repositories* (2014), 22–31.

[148] Pinto, L.S. et al. 2013. TestEvol: A tool for analyzing test-suite evolution. *Proceedings - international conference on software engineering* (2013), 1303–1306.

[149] Pinto, L.S. et al. 2012. Understanding myths and realities of test-suite evolution. *Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering* (2012), 33.

[150] Plewnia, C. et al. 2014. On the influence of release engineering on software reputation. *Mountain view, ca, usa: In 2nd international workshop on release engineering* (2014).

[151] Poo-Caamaño, G. 2016. *Release management in free and open source software ecosystems*.

[152] Radjenović, D. et al. 2013. Software fault prediction metrics. *Information and Software Technology*. 55, 8 (Aug. 2013), 1397–1418.

[153] Raemaekers, S. et al. 2017. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*. 129, (Jul. 2017), 140–158.

[154] Rahman, F. and Devanbu, P. 2013. How, and why, process metrics are better. *2013 35th international conference on software engineering (icse)* (May 2013), 432–441.

[155] Rahman, F. et al. 2011. BugCache for inspections: Hit or miss? *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (New York, NY, USA, 2011), 322–331.

[156] Rajlich, V. 2014. Software evolution and maintenance. *Proceedings of the on future of software engineering - FOSE 2014* (2014).

[157] Rausch, T. et al. 2017. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. *Proceedings of the 14th international conference on mining software repositories* (2017), 345–355.

[158] Robbes, R. et al. 2012. How do developers react to API deprecation? *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering - FSE 12* (2012).

[159] Robinson, B. et al. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. *2011 26th ieee/acm international conference on automated software engineering (ase 2011)* (Nov. 2011), 23–32.

[160] Rodríguez, P. et al. 2017. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*. 123, (2017), 263–291.

[161] Romano, S. et al. 2017. Findings from a multi-method study on test-driven development. *Information and Software Technology*. 89, (2017), 64–77.

[162] Saborido, R. et al. 2018. An app performance optimization advisor for mobile device app marketplaces. *Sustainable Computing: Informatics and Systems*. (2018).

[163] Santolucito, M. et al. 2018. Statically verifying continuous integration configurations. *arXiv preprint*

arXiv:1805.04473. (2018).

[164] Schneidewind, N.F. 2007. Risk-driven software testing and reliability. *International Journal of Reliability, Quality and Safety Engineering*. 14, 2 (2007), 99–132.

[165] Scoccia, G.L. et al. 2018. An investigation into android run-time permissions from the end users' perspective. (2018).

[166] Shamshiri, S. et al. 2018. How do automatically generated unit tests influence software maintenance? *Software testing, verification and validation (icst), 2018 ieee 11th international conference on* (2018), 250–261.

[167] Shepperd, M. et al. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*. 40, 6 (June 2014), 603–616.

[168] Shimagaki, J. et al. 2016. A study of the quality-impacting practices of modern code review at sony mobile. *Software engineering companion (icse-c), ieee/acm international conference on* (2016), 212–221.

[169] Souza, R. et al. 2015. Rapid releases and patch backouts: A software analytics approach. *IEEE Software*. 32, 2 (2015), 89–96.

[170] Stallman, R. 2002. *Free software, free society: Selected essays of richard m. stallman*. Lulu. com.

[171] State of the union: Npm: 2016. *https://www.linux.com/news/event/Nodejs/2016/state-union-npm*. Accessed: 2018-10-11.

[172] Statista 2018. Number of apps available in leading app stores as of 1st quarter 2018. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores.

[173] Stolberg, S. 2009. Enabling agile testing through continuous integration. *Agile conference, 2009. agile'09.* (2009), 369–374.

[174] Teixeira, J. 2017. Release early, release often and release on time. an empirical case study of release management. *Open source systems: Towards robust practices* (Cham, 2017), 167–181.

[175] Teixeira, J. et al. 2015. Lessons learned from applying social network analysis on an industrial free/libre/open source software ecosystem. *Journal of Internet Services and Applications*. 6, 1 (Jul. 2015).

[176] The npm blog: Kik, left-pad and npm: 2016. *https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm*. Accessed: 2018-10-15.

[177] The redmonk programming language rankings: January 2018: 2018. *https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/*. Accessed: 2018-10-11.

[178] Thongtanunam, P. et al. 2017. Review participation in modern code review. *Empirical Software Engineering*. 22, 2 (2017), 768–817.

[179] Thongtanunam, P. et al. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. *Proceedings of the 38th international conference on software engineering* (2016), 1039–1050.

[180] Thongtanunam, P. et al. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. *Software analysis, evolution and reengineering (saner), 2015 ieee 22nd international conference on* (2015), 141–150.

[181] Thongtanunam, P. et al. 2014. Reda: A web-based visualization tool for analyzing modern code review dataset. *Software maintenance and evolution (icsme), 2014 ieee international conference on* (2014), 605–608.

[182] Trockman, A. 2018. Adding sparkle to social coding. *Proceedings of the 40th international conference on software engineering companion proceeedings - ICSE 18* (2018).

[183] Vasilescu, B. et al. 2014. Continuous integration in a social-coding world: Empirical evidence from

github. *Software maintenance and evolution (icsme), 2014 ieee international conference on* (2014), 401–405.

[184] Vassallo, C. et al. 2018. Un-break my build: Assisting developers with build repair hints. (2018).

[185] Vassallo, C. et al. 2017. A tale of ci build failures: An open source and a financial organization perspective. *Software maintenance and evolution (icsme), 2017 ieee international conference on* (2017), 183–193.

[186] Vernotte, A. et al. 2015. *Risk-driven vulnerability testing: Results from eHealth experiments using patterns and model-based approach*.

[187] Wang, H. et al. 2018. Why are android apps removed from google play?: A large-scale empirical study. *Proceedings of the 15th international conference on mining software repositories* (2018), 231–242.

[188] Wang, S. et al. 2016. Automatically learning semantic features for defect prediction. *2016 ieee/acm 38th international conference on software engineering (icse)* (May 2016), 297–308.

[189] Wei, L. et al. 2017. OASIS: Prioritizing static analysis warnings for android apps based on app user reviews. *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (2017), 672–682.

[190] Widder, D.G. et al. 2018. I'im leaving you, travis: A continuous integration breakup story. (2018).

[191] Xie, Z. et al. 2016. You can promote, but you can't hide: Large-scale abused app detection in mobile app stores. *Proceedings of the 32nd annual conference on computer security applications* (2016), 374–385.

[192] Yang, X. et al. 2016. Mining the modern code review repositories: A dataset of people, process and product. *Proceedings of the 13th international conference on mining software repositories* (2016), 460–463.

[193] Zaidman, A. et al. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*. 16, 3 (2011), 325–364.

[194] Zampetti, F. et al. 2017. How open source projects use static code analysis tools in continuous integration pipelines. *Mining software repositories (msr), 2017 ieee/acm 14th international conference on* (2017), 334–344.

[195] Zanjani, M.B. et al. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*. 42, 6 (2016), 530–543.

[196] Zhao, Y. et al. 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. *Proceedings of the 32nd ieee/acm international conference on automated software engineering* (2017), 60–71.

[197] Zimmermann, T. et al. 2009. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. *Proceedings of the the 7th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering* (New York, NY, USA, 2009), 91–100.