QUALITY ASSURANCE IN RESEARCH SOFTWARE

by

NASIR U. EISTY

JEFFREY CARVER, COMMITTEE CHAIR
JEFF GRAY
ZHE JIANG
DINGWEN TAO
HAI AH NAM

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2020

Copyright Nasir U. Eisty 2020
ALL RIGHTS RESERVED

ABSTRACT

Breakthroughs in research increasingly depend on complex software libraries, tools, and applications aimed at supporting specific science, engineering, business, or humanities disciplines. Collectively, we call these software, libraries, tools, and applications as research software. Research software plays an important role in solving real-life problems, scientific innovations, and handling emergency situations. So the correctness and trustworthiness of research software are of absolute importance. The complexity and criticality of this software motivate the need for proper software quality assurance through different software engineering practices. Software metrics, software development process, peer code review and software testing are four key tools for assessing, measuring, and ensuring software quality and reliability.

The goal of this dissertation is to better understand how research software developers use traditional software engineering concepts of software quality to support and evaluate both the software and the software development process. One key aspect of this goal is to identify how the four quality practices relevant to research software corresponds to the practices commonly used in traditional software engineering. I used empirical software engineering research methods to study the human aspects related to using software quality practices for the development of research software. I collected information related to the four software activities through surveys, interviews, and directly working with research software developers. Research software developers appear to be interested and see value in software quality practices, but may be encountering roadblocks when trying to use them. Through this dissertation, beside current practices, I identified challenges and improvements of the current practices.

DEDICATION

To my parents for their love, endless support, and encouragement.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Jeffrey Carver, for his endless support, direction, and feedback throughout my Ph.D. program. His dynamism, vision, sincerity, and motivation have deeply inspired me. It was a great privilege and honor to work and study under his guidance. I am extremely grateful for what he has offered me. I am very much thankful to Dr. Jeff Gray for his help in many ways in my Ph.D. life. I would like to thank the rest of the committee members: Dr. Hai Ah Nam, Dr. Dingwen Tao, and Dr. Zhe Jiang for their encouragement, insightful comments, and guidance in helping me complete the dissertation. I also gratefully thank Dr. David Bernholdt from Oak Ridge National Laboratory, Dr. George Thiruvathukal from Loyola University Chicago, Dr. Danny Perez and Dr. J. Dave Moulton from Los Alamos National Laboratory, and Dr. Roland Haas, Dr. Gabrielle Allen, and Dr. Daniel Katz from National Center for Supercomputing Applications for their contributions to this research.

I thank all my fellow labmates from the Software Engineering research group of the University of Alabama for providing valuable feedback and suggestions and for offering me plentiful opportunities to present my research. I would especially like to thank them for their support and good friendship - we had great and brilliant times together. I am extremely grateful to my wife, Moumita Tabassum, for her patience, encouragement and always being on my side. Last but not least, I acknowledge support from the NSF grant 1445344.

CONTENTS

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER 1 INTRODUCTION 1

1.1 Research Goals 3

1.2 Plan of Action 3

1.3 Expected Contributions 6

CHAPTER 2 SOFTWARE METRICS AND SOFTWARE DEVELOPMENT PROCESS IN RESEARCH SOFTWARE 7

2.1 Introduction 7

2.2 Research Questions 9

2.2.1 Metrics 9

2.2.2 Code Complexity 12

2.2.3 Process 13

2.3 Survey Design 15

2.4 Results 16

2.4.1 Demographics 16

2.4.2 RQ1: What is the overall level of metrics knowledge and use by research software developers? 20

2.4.3 RQ2: What is the relationship between knowledge of metrics and their perceived usefulness? . . . 21

2.4.4 RQ3: Which metrics are most commonly used? . . . 22

2.4.5 RQ4: To what extent do the demographics of the respondents affect their perception and use of software metrics? . . . 26

2.4.6 RQ5: Do research software developers perceive code complexity as a problem? . . . 30

2.4.7 RQ6: Is the frequency of complexity problems related to the use of or perceived helpfulness of metrics? . . . 31

2.4.8 RQ7: To what extent do research software developers follow software process? . . . 32

2.4.9 RQ8: What value do research software developers place on software process? . . . 34

2.4.10 RQ9: Is there any relationship between the valued placed on software process (RQ8) and the use of software process (RQ7)? . . . 34

2.4.11 RQ10: Which software development processes do research software teams use? . . . 35

2.4.12 RQ11: To what extent do research software teams use software process metrics? . . . 36

2.4.13 RQ12: To what extent do the demographics of the respondents affect their perception and use of software process? . . . 36

2.5 Threats . . . 40

2.5.1 Internal Threats . . . 40

2.5.2 External Threats . . . 41

2.5.3 Construct Threats . . . 41

2.6 Conclusion . . . 42

CHAPTER 3 PEER CODE REVIEW IN RESEARCH SOFTWARE . . . 44

3.1 Introduction . . . 44

3.2 Background and Motivation . . . 47

3.3 Research Questions 49

3.4 Methodology 50

3.4.1 Survey/Interview Design 50

3.4.2 Data Collection 51

3.4.3 Data Analysis 52

3.5 Results 55

3.5.1 Demographics 55

3.5.2 RQ1: How do research software developers perform code review? . . . 56

3.5.3 RQ2: What effect does code review have on research software? 65

3.5.4 RQ3: What difficulties do research software developers face with code review? 67

3.5.5 RQ4: What improvements to the code review process do research software developers need? 69

3.6 Discussion 70

3.7 Threats 71

3.7.1 Internal Threats 71

3.7.2 External Threats 72

3.7.3 Construct Threats 72

3.8 Conclusion 72

CHAPTER 4 A CASE STUDY OF TESTING RESEARCH SOFTWARE 74

4.1 Introduction 74

4.2 Background 75

4.2.1 ParSplice 76

4.2.2 Metamorphic Testing 77

4.2.3 Run-time Assertion Checking 77

4.2.4 Machine Learning Techniques 78

4.3 Case Study . . . 78
4.3.1 PSIP . . . 78
4.3.2 Multinomial Test . . . 79
4.3.3 Results . . . 80
4.4 Conclusion . . . 81
CHAPTER 5 A SURVEY OF TESTING RESEARCH SOFTWARE . . . 83
5.1 Introduction . . . 83
5.2 Research Questions . . . 85
5.3 Methodology . . . 88
5.3.1 Survey Design . . . 88
5.3.2 Data Analysis . . . 89
5.4 Results . . . 92
5.4.1 Demographics . . . 92
5.4.2 RQ1: What level of knowledge do research software developers have about testing techniques? . . . 94
5.4.3 RQ2: How do research software developers test their software? . . . 97
5.4.4 RQ3: Why is testing research software difficult? . . . 100
5.4.5 RQ4: Is it possible to adapt existing testing techniques to support the testing of research software? . . . 103
5.4.6 RQ5: What improvements to the testing process do research software developers need? . . . 107
5.5 Discussion . . . 110
5.5.1 RQ1 - Knowledge . . . 110
5.5.2 RQ2 - Practices . . . 110
5.5.3 RQ3 - Difficulties . . . 111
5.5.4 RQ4 - Adapting existing testing methods . . . 111

5.5.5 RQ5 - Improvement 112
5.6 Threats 112
5.6.1 Internal Threats 112
5.6.2 External Threats 113
5.6.3 Construct Threats 113
5.7 Conclusion 113
CHAPTER 6 CONCLUSION 115
6.1 Summary 115
6.2 Contributions 116
6.2.1 Metrics 116
6.2.2 Process 117
6.2.3 Code Review 118
6.2.4 Testing 118
6.3 Recommendations 119
6.3.1 Metrics and Process 119
6.3.2 Code Review 120
6.3.3 Testing 121
6.4 Future work 122
REFERENCES 124
APPENDIX A SOFTWARE METRICS IN RESEARCH SOFTWARE 132
APPENDIX B TESTING TECHNIQUES IN RESEARCH SOFTWARE 134
APPENDIX C INSTITUTIONAL REVIEW BOARD CERTIFICATION 136

LIST OF TABLES

2.1 Perceived Usefulness of Metrics vs. Knowledge of Metrics 22

2.2 Categories of Metrics Used 24

2.3 Perceived Value of Process vs. Use of Process 35

5.1 List of Used Testing Techniques. 99

LIST OF FIGURES

2.1 Survey Questions Part I 17

2.2 Survey Questions Part II 18

2.3 Number of Developers 19

2.4 Respondents' Role on Project 19

2.5 Project Stage 20

2.6 Knowledge of Metrics 21

2.7 Perceived Usefulness of Metrics 21

2.8 Frequency of using Metrics for Evaluation 25

2.9 Influence of project size on overall knowledge of metrics 27

2.10 Influence of project size on perceived usefulness of metrics 27

2.11 Influence of project size on known metrics 28

2.12 Influence of project size on used metrics 28

2.13 Influence of project stage on overall knowledge 29

2.14 Influence of project stage on perceived usefulness 30

2.15 Influence of project stage on specific metrics used 30

2.16 Frequency of Code Complexity Problems 31

2.17 Use of Code Complexity Metrics 31

2.18 Usefulness of Code Complexity Metrics 32

2.19 Frequency of following process . . . 33

2.20 Frequency of value seen in process use . . . 34

2.21 Frequency of using particular process . . . 36

2.22 Frequency of process metrics use . . . 37

2.23 Influence of project size on process following as a team . . . 38

2.24 Influence of project size on personally following process . . . 38

2.25 Influence of project size on value seen . . . 38

2.26 Influence of project stage on process following . . . 39

2.27 Influence of project stage on value seen . . . 40

3.1 Survey Questions Part I . . . 53

3.2 Survey Questions Part II . . . 54

3.3 Number of years worked on research software . . . 56

3.4 Respondents' role on project . . . 56

3.5 Balance as a reviewee and reviewer . . . 57

3.6 Percentage of code undergo review . . . 58

3.7 Number of reviewers . . . 58

3.8 Factors to accept code review request . . . 59

3.9 Time spent on code review . . . 60

3.10 Time for a first response . . . 61

3.11 Time for a final decision . . . 62

3.12 Problems code review identifies . . . 62

3.13 Positive experiences with code review . . . 63

3.14 Negative experiences with code review . . . 64

3.15 Explanation of why code review is important 65
3.16 Explanation of why code review helps improve the code 66
3.17 Code review helps decrease code complexity 67
3.18 Challenging aspects of code review 68
3.19 Barriers reviewers face when reviewing code 69
3.20 Improvement areas in the code review process 70
4.1 p-values obtained by executing *ParSplice* for different times. 81
5.1 Survey Questions Part I 90
5.2 Survey Questions Part II 91
5.3 Respondents' role on project 93
5.4 Number of years worked on research software 93
5.5 Project stage 94
5.6 Number of developers 95
5.7 Confidence on knowledge of software testing 95
5.8 Level of understanding on the testing concepts used 96
5.9 Level of understanding on the testing concepts needed 96
5.10 Goal of testing 98
5.11 Testing methods used 100
5.12 Usefulness of testing 100
5.13 Complexity of testing research software 101
5.14 Challenges in testing research software 103
5.15 Applying Commercial/IT testing methods by team 104
5.16 Applying Commercial/IT testing methods personally 104

5.17 Value seen in using Commercial/IT testing methods 105

5.18 Challenges to adapt Commercial/IT testing methods 106

5.19 Challenges could not met by Commercial/IT testing methods 107

5.20 Improvement of the testing methods 109

CHAPTER 1

INTRODUCTION

Researchers in a number of scientific, engineering, business, and humanities domains increasingly develop and/or use software to conduct or support their research. Without such software, it would be difficult or impossible for many researchers to do their work. Collectively, we refer to the software (libraries, tools, and applications) developed by these researchers as *research software*. Research software includes both software for end-user researchers and software that provides infrastructure support, including messaging middleware, scheduling software, and various mathematical, scientific, or statistical libraries.

Results produced by this software contribute to the prediction of natural phenomena, and support decisions about the world's critical needs. Low quality software is likely to produce less trustworthy results, may lead to incorrect research conclusions, and is prone to failure in mission-critical situations. Therefore, because the correctness of the design and implementation of this software is very important, researchers need proper software engineering practices to produce correct results and develop high-quality research software.

We have observed that research software developers often place less importance on traditional views of software quality and maintainability than on other scientific goals. The characteristics of research software and of the research software developer lead them to be more concerned about the underlying scientific goal than about the use of appropriate software engineering practices. There have been many efforts to understand how software engineering can help with the development and maintenance of research software,

especially related to the development process (including requirements engineering, design methods, and testing) and to code complexity (including refactoring). However, there is little evidence that these efforts have led research software teams to value software process in a manner similar to how more traditional software teams value software process.

Data dependencies within the software must have some influences in the design phase of the research software development lifecycle. During the software design phase, research software developers also need to identify some appropriate strategies like parallelization of the scientific algorithms, fault tolerance mechanisms, and parallel computations. The presence of complex communication or I/O patterns could also degrade the performance of research software. The coding phase requires a highly specialized skill set in numerical algorithms and systems to squeeze out performance. The validation and verification phase for research software development differs remarkably from traditional systems because the results are somewhat unpredictable for new science or algorithms. There are also some architectural issues with the use of popular software engineering tools in the research domain. Even in the deployment phase, larger node and core sizes, coupled with long runtimes, result in an increased likelihood of failure of computing elements.

In addition to the challenges presented by these methodological differences, research software development also faces people-related challenges. First, the lack of interdisciplinary computational science programs reduces the pipeline of graduates who have the experience required to be effective in the research software domain. Second, the lack of these programs reduces the motivation for graduates to pursue careers in research software. Third, the knowledge, skills, and incentives present in research software development differ from those present in traditional software domains. For example, research software developers may lack formal software engineering training, trained software engineers may lack the required depth of understanding of the science domain, and the incentives in the research domain focus of timely results, rather than more traditional software quality/productivity goals.

1.1 Research Goals

By investigating research software developers, I hope to illuminate problem areas of software quality within the community. Identifying and documenting the problems will help motivate research software developers to use different software quality practices and produce higher-quality research software. By reviewing the literature and directly working with research software developers, I identified four key software quality practices: metrics, software development process, peer code review, and software testing. Because these practices are essential for any kind of software development and there is a lack of proper use in research software projects, this dissertation focuses on them. Therefore, the overall goal of this dissertation is to *investigate the use of different software quality practices including motivations, challenges, barriers, and improvements in research software development.*

Once I have identified the specific problems related to these practices, I will suggest solutions to help research software developers move forward and produce high-quality research software. Therefore, the second goal of this dissertation is to *provide recommendations on how to move forward by solving software quality problems in the research software community.*

1.2 Plan of Action

This is an article-style dissertation where each article builds a piece of the argument. To produce high-quality and reliable research software, research software developers must follow a proper software development process, use valid metrics, and employ appropriate software quality activities, such as testing and code review. I describe all these measures and activities in four distinct articles. Combining the articles together, this dissertation accomplishes the overall goal of quality assurance in research software development. The impact should lead research software developers to both short-term benefits through improved results and long-term benefits through more maintainable software.

Article 1 begins by investigating the use of software metrics and software development processes. A software metric describes an essential property of software quality and is used in measuring quantifiable characteristics. Software development processes divide the development work to improve the design, product management, and project management. The overall goal of this article is to *better understand how research software developers use traditional software engineering concepts, like metrics and processes, in their projects.*

The primary contributions of this article are:

- An overview of the use of metrics by research software developers;
- Identification of software metrics of interest and value to research software developers.
- The perceived prevalence of code complexity and whether code complexity metrics are used to manage it.
- An overview of how research software developers use and perceive the importance of software process.
- Identification of the software processes preferred by research software developers.
- The perceived prevalence of software process and whether it is useful to research software developers.

Article 2 describes a software quality assurance activity, peer code review. Peer code review is manually inspecting source code by one or several peers. Because peer code review is less prevalent in research software, compared with other types of software, the goal of this work is to *understand current practice of peer code review in the development of research software, identify challenges and barriers associated with peer code review in research software, and present approaches to improve the peer code review in research software.*

The key contributions of this article are:

- An overview of the current code review practices in research software development.
- Identification of the defects identified during the code review process.
- Positive and negative experiences research software developers have regarding code review.
- Impacts of the the code review process in the project.
- Challenges and barriers developers face during code review.
- Potential areas of improvement in the review process.

Article 3 discusses another software quality assurance activity, testing, in research software development. This article describes a case study of developing testing infrastructure for a research software project by using a statistical test. The overall goal of this article is to *demonstrate the use of a statistical method for testing research software.*

The key contributions of this article are:

- Implementation of testing infrastructure of a non-deterministic parallel research software.
- An overview of available testing techniques to test non-deterministic stochastic research software.
- Demonstration of the use of a statistical testing method to test research software.

The challenges during the work in article 3 motivates me to learn more about the testing practices in the research software community. I developed an online survey that illustrates the overall use of software testing techniques in research software development. Article 4 describes the results of the survey and provides an overview of the testing process in the research software community. The overall goal of this article is to *better understand the difficulties of testing research software and identify potential ways to improve the current testing practices.*

The key contributions of this article are:

- An overview of the level of knowledge research software developers have on testing techniques;
- A description of the current testing practices used in the research software community;
- A list of the difficulties of the testing research software;
- An analysis of the compatibility of commercial/IT testing techniques to research software; and
- An identification of areas of improvement in the testing process for research software.

1.3 Expected Contributions

Overall, this dissertation will make several contributions to the literature, including:

- An overview of the use of different software metrics and software development process in research software development.
- Investigation of the peer code review practices such as challenges, barriers, and improvement opportunities in research software development.
- Determine software testing best practices in research a software setting including software testing knowledge and use among the research software developers, difficulties to test research software, and potential ways of improvement of the testing infrastructure.

CHAPTER 2

SOFTWARE METRICS AND SOFTWARE DEVELOPMENT PROCESS IN RESEARCH SOFTWARE

2.1 Introduction

Software metrics are a critical tool for building reliable software and assessing software quality, especially in complex domains and/or mission-critical environments. The ultimate goal of software metrics is to provide continuous insight into products and processes. A useful metric typically performs a calculation to assess the effectiveness of the underlying software or process. The established literature distinguishes individual *measures* from *metrics*. A metric is a *function*, whereas a *measurement* is the application of metrics to obtain a value. A detailed description of metrics is beyond the scope of this chapter. We refer readers to authoritative texts on the subject [25].

Software process is a collection of activities, actions, and tasks performed during the creation of a software work product [65] that has the goal of creating *high quality software*. Software process, therefore, focuses on the work product of *software* with the ultimate goal of creating a *high quality software*. There exist many software process models: waterfall, spiral, iterative, and agile (being one of the most recent), just to name the major ones. With the exception of agile methods, the ones commonly associated with large-scale software development are waterfall, spiral, and iterative.

Our experiences working with research software developers who claim to embrace software development process suggest the importance of two general classes of metrics: in-process (related to development process) and code-oriented (primarily code complexity).

Furthermore, our cursory analysis of the landscape of research software, much of which is open source, confirms that many aspects of process are present in these projects (e.g., version control, issue tracking, testing, and documentation).

Software process is of critical importance in building reliable software and software metrics is an important software quality tool, in this article we look beyond our anecdotal experiences in order to attain a deeper understanding of perceptions about software metrics and process directly from research software developers. The primary objective of this study is to *understand research software developers' knowledge and use of software metrics and gain a better understanding of the use of software development process for research software in support of the development process.*

To gather this information, we developed and distributed a survey to research software developers. The survey provided respondents an opportunity to provide feedback on the impact various types of software metrics and software development process have had on their respective projects.

The primary contributions of this article are:

- An overview of the use of metrics by research software developers.
- Identification of software metrics of interest and value to research software developers.
- The perceived prevalence of code complexity and whether code complexity metrics are used to manage it.
- An overview of how research software developers use and perceive the importance of software process.
- Identification of the software processes preferred by research software developers.
- The perceived prevalence of software process and whether it is useful to research software developers.

The remainder of this paper is organized as follows. In Section 2.2 we describe previous work to motivate a series of research questions explored in this study. In Section 2.3 we explain the survey design. In Section 2.4 we provide the detailed survey results. In Section 2.5 we enumerate the validity threats. In Section 2.6 we draw conclusions.

2.2 Research Questions

In this section, we define our research questions based upon a discussion of the related work. These research questions drive the survey design.

2.2.1 Metrics

Research software developers often have a general interest in metrics, including some that are not directly related to software development. The literature describes a number of these non-traditional metrics along with why those metrics are important in the research software domain.

First, *performance* (speed of execution) is critical for a segment of the research software developer population. Therefore, it is not uncommon for research software developers to have an interest in measures like FLOPS (floating point operations/second) or I/O (reads or writes per second) and network throughput (MB/second). The Top500 list[1], which ranks the performance of supercomputers while executing a common benchmark, is an example of a research software metric that is not common in traditional SE environments. Even though the benchmark does not cover all aspects of performance, the Top500 is an example of a metric that the community perceives to be useful.

Second, beyond performance, many research software developers have a relatively new interest in *green computing* [31]. This interest increases the relevance of concepts like energy costs and sustainability. Specifically, some research software developers focus on *energy efficiency* and *carbon emissions* [49].

Third, in some sub-disciplines of research software (e.g., simulation and modeling),

[1] `http://top500.org`

developers find *correctness* and *reproducibility* important. The lack of these characteristics can decrease the "velocity of science." There is a need for metrics that allow developers to describe their results, along with the acceptable level of error tolerance, so that other researchers can reproduce the results [75].

Fourth, the *failure rate* of software, that is frequency with which the software fails to produce a correct answer (or to produce an answer at all), is critical for some types of research software. For example, it is important for research software developers to measure and understand the failure rate when the software is targeted at very large (i.e., 10,000 node) computers [68]. Similarly, in computer vision software (a subcategory of research software), it is important for research software developers to measure how often their algorithms fail to reach the correct decision for a given image [76].

Finally, research software developers (discussed in the introduction) seek *recognition* for their work on defining and developing research software. Baxter et al. describe two ends of the research software spectrum that seek recognition: (1) researcher-developers, who want to be judged on their scientific output but are mostly producing code in support of research and (2) research software developers, who not only produce code but produce tools that help others to do research [5].

Although these metrics are different than the metrics traditionally found in the general SE literature, they are highly relevant to many research software developers. Therefore, they help to inform our overall understanding of the types of metrics that research software developers perceive to be useful.

Given the fact that we were not able to identify many papers that discuss the use of traditional software metrics, prior work by Carver and Heaton also helps to inform our research [10]. Although developers indicated they had sufficient knowledge to do their jobs, a survey of research practitioners (from this same work) revealed some interesting findings about software engineering knowledge within the research community.

- Most research practitioners have little formal SE training and tend to be self-taught.

- One-third of the respondents thought that overall the research communities' SE skills were not adequate.
- Familiarity of SE methods was higher than use of those methods.
- Code reviews and agile methods were rarely used, suggesting a lack of collaborative development practices.
- The knowledge and perceived relevance of agile methods was low.

The last observation is interesting because agile practices most closely resemble how the typical research team operates. While these findings are about general SE and SE process, they inform the study of SE metrics by suggesting that research software developers may perceive they have significant knowledge of metrics, but may not use that knowledge as frequently. Similarly, Carver and Heaton's systematic review found a number of claims made about the use of software engineering practices in the development of research software [32] including:

- There is a limited use of testing;
- Many research teams embrace (perhaps unconsciously) an agile mode of development.

With this background to understand metrics usage and SE knowledge within research software teams, we pose the following research questions to gain a better understanding of the knowledge and use of metrics by research software developers:

- ***RQ1*** – *What is the overall level of metrics knowledge and use by research software developers?*
- ***RQ2*** – *What is the relationship between knowledge of metrics and their perceived usefulness?*
- ***RQ3*** – *Which metrics are most commonly used?*

- ***RQ4*** – *To what extent do the demographics of the respondents affect their perception and use of software metrics?*

2.2.2 Code Complexity

Beyond general metrics, research software developers are interested in gaining an understanding of code complexity, which is a nagging problem in research software. In our experience, research software often contains innate complexity. In addition, developers often introduce additional complexity into research software through various compiler pragmas, concurrent/parallel language features, and/or programming libraries for code optimization. A good example of this complexity is parallel matrix multiplication [62], a common assignment in university courses. The algorithm is one of the relatively straightforward research examples when written for sequential processing. When scaling up to multiprocessors and clustered systems, however, it requires much more complex code, sometimes of an architecture-specific nature.

First, *code complexity* is critical for a segment of the research software developer population that focuses on trying to make code run on parallel architectures. For example, Munipala et al. explored the use of SLOC and cyclomatic complexity in a collection of diverse GPGPU software packages (a subset of the research software community). They used the commercial McCabe IQ tool to measure cyclomatic, design, and essential complexity metrics in the software packages. While the results of the study were inconclusive about whether SLOC or complexity metrics were more prominent, the tools helped identify potentially large and/or complex modules [53]. This example shows that there is at least some interest in the research software community for applying *code complexity tools* that perform static analysis to identify potential complexity issues.

Second, *module size* is critical for a segment of the research population. In another case study on complex open source software, aimed at understanding both the implications of structural quality and the benefits of structural quality analysis, Stamelos et al. found that the average component size (a dimension of code complexity but a separate metric) of

an application is negatively related to user satisfaction [72]. While this particular paper did not focus directly on research software, it is relevant to our work, because many research projects are released as open source and are both large and complex in nature. Similar to Munipala et al.'s work, which also included module size, this result shows that both research and open source communities have concerns about code complexity.

Based on this discussion, we pose the following research questions to gain a better understanding of whether code complexity is encountered by research software developers:

- ***RQ5*** – *Do research software developers perceive code complexity as a problem?*
- ***RQ6*** – *Is the frequency of complexity problems related to the use of or perceived helpfulness of metrics?*

2.2.3 Process

The literature shows that researchers in a number of science and engineering domains increasingly use software development processes. The reports from these domains, which include physics [28], astrophysics [18], and biomedical imaging [13], reveal a range of goals for software process. The goals include developing modular and extensible software to developing software that is open for user contributions. Research software developers in the biomedical imaging domain specifically use software process to help improve interactions between domain scientists and developers [13].

At least one large-scale effort highlights the continued value of traditional software process for research software [59]. The ASCI project, a project aimed at reducing the nuclear stockpile through simulation and modeling, made use of traditional software development processes. The ASCI project found that techniques associated with traditional process to be of value, including, but are not limited to, estimation, resource management, team expertise/skill sets, risk analysis, working with stakeholders, training, and verification and validation.

These observations lead to the next three research questions:

- ***RQ7*** – *To what extent do research software developers follow software process?*
- ***RQ8*** – *What value do research software developers place on software process?*
- ***RQ9*** – *Is there any relationship between the value placed on software process (RQ8) and the use of software process (RQ7)?*

With respect to which particular software development process research software teams follow, the literature is mixed. A systematic literature review found that many research teams use agile approaches and variants thereof, even if they are unaware [32]. A review of *agile practices* [71] in scientific software development mapped the use of prominent agile reference models (e.g. Scrum and eXtreme Programming, a.k.a. XP, etc.). A key finding of this mapping study is that scientific software development projects not only adopt agile practices but also perceive their testing to be better than average.

Therefore, to better understand the types of software processes research software teams use the next research question is:

- ***RQ10*** – *Which software development processes do research software teams use?*

Research software developers will benefit from less intrusive processes [50]. To determine whether a particular software process is effective, the team must conduct measurement and analysis. The authors of this study propose collecting data for measuring process effectiveness to decide what practices might be most effective. Although there has been little follow-up to validate this approach, we find it positive that a research software team believe in the importance of measuring process with an eye towards improving it. Therefore, the next research question is:

- ***RQ11*** – *To what extent do research software teams use software process metrics?*

Lastly, a survey of research practitioners revealed some interesting findings about *software engineering knowledge* within the research community [11]:

- Most research practitioners have little formal SE training and tend to be self-taught.
- One-third of the respondents thought that overall the research communities' SE skills were not adequate.
- Familiarity of SE methods was higher than the use of those methods.
- Code reviews and agile methods were rarely used, suggesting a lack of collaborative development practices.
- The knowledge and perceived relevance of agile methods were low.

There is some inconsistency in the level of knowledge and understanding research software developers have about software development process. Because research software developers often have different backgrounds than traditional software engineers, various demographics could affect their level of knowledge about SE. To better understand whether these factors also affect research software developers, we pose the following research question:

- ***RQ12*** – *To what extent do the demographics of the respondents affect their perception and use of software process?*

2.3 Survey Design

To answer each research question defined in Section 2.2, we enumerated a series of survey questions. We took care in writing the questions to ensure their wording did not bias the respondents. For example, we asked a free-response question about metrics rather than providing a pre-determined list. We grouped the survey questions by topics to help respondents focus. Figure 2.1 and 2.2 show the survey questions that we include in this analysis. The figures contain the questions along with the possible answer choices for each question (note '[free response]' indicates a free-response question).

To reach a broad audience of research software domains we used three solicitation methods. First, we sent the survey invitation to a series of mailing lists that target

developers and users of mathematical, science, and engineering software. Those mailing lists included: hpc-announce@mcs.anl.gov (a mailing list based at Argonne National Laboratory that targets researchers who use high-performance computing in their work), the PI list for the US National Science Foundation SI2 (Software Infrastructure for Sustained Innovation) PI mailing list, and Carver's list of previous participants in the SE4Science workshop series (`http://www.SE4Science.org/workshops`). Second, a collaborator sent the survey to the mailing list of Research Software Engineers in the UK. Third, we advertised the survey in a column in *Computing in Science & Engineering* [12] where many research software developers/practitioners would be likely to see it. In both cases, we also asked people to forward the survey invitation within their own networks. As a result of our solicitation approach, we are not able to estimate the number of people who received the invitation.

2.4 Results

In this section, we present the results of the survey organized around the survey themes. we included a response in the analysis if the respondent answered at least 90% of the questions. In total, we received 129 responses to the survey. Note that throughout this discussion, the survey questions refer to the numbers in Figure 2.1 and 2.2.

2.4.1 Demographics

For each demographic, we give a brief explanation for why the demographic is relevant and any implications the demographic has for the survey analysis. We use these demographics in the next subsection to better understand the overall results. The discussion of results is based upon the data from the respondents who completed the survey.

Project Types

The goal of this demographic is to understand whether we reached the target audience. Question GQ1 lists the possible responses. In the actual survey, we provided

Figure 2.1: Survey Questions Part I

General Questions

GQ1 Which of the following best describes your project? [Scientific Computing Software, Computer Science Software, General Application Software, Other]

GQ2 How many FTEs of developers are currently on your project? [free-response]

GQ3 Which best describes your role on the project? [Developer, Architect, Manager, Executive, Other]

GQ4 Which best describes the current development stage of your project? [Planning/Requirements Gathering, Initial Development/Prototyping, Active Development/Unreleased Software, Active Development/Released Software, Maintenance/No New Development Planned, Other]

Metrics Questions

MQ1 What is your level of knowledge about software metrics in general? [Very Low, Low, Average, High, Very High]

MQ2 List any software metrics with which you are familiar [free-response]

MQ3 How often are software metrics useful to your project? [Very Low, Low, Average, High, Very High]

MQ4 Which specific software metrics are most useful to your project? [free-response]

MQ5 How often are software metrics used to evaluate individual productivity on your team? [Never, Rarely, Sometimes, Most of the Time, Always]

MQ6 How often are software metrics used in the aggregate to evaluate overall team productivity? [Never, Rarely, Sometimes, Most of the Time, Always]

Code Complexity Questions

CQ1 How often is code complexity a problem in your software? [Never, Rarely, Sometimes, Most of the Time, Always]

CQ2 How frequently does your team use code complexity metrics? [Never, Rarely, Sometimes, Most of the Time, Always]

CQ3 How often do code complexity metrics actually help your team to understand and reduce code complexity? [Never, Rarely, Sometimes, Most of the Time, Always]

CQ4 Which specific code complexity metrics do you use? [free-response]

examples to clarify the meaning of each option. The fact that majority of the respondents indicated that they worked on *Scientific Computing Software* indicates that the survey did

Figure 2.2: Survey Questions Part II

Process Questions

PQ1 How often does your team follow a defined software development process? [Never, Rarely, Sometimes, Most of the Time, Always]

PQ2 How often do you personally follow a defined software development process? [Never, Rarely, Sometimes, Most of the Time, Always]

PQ3 How much value do you see personally in following a defined software development process? [Very Little, Little, Moderate, High, Very High]

PQ4 How often does your team use each of the following software development processes? [Agile, Rapid Application Development, Ad Hoc, Waterfall model, Spiral model, Use-Case Methodology]

PQ5 How often does your team use process metrics to evaluate the effectiveness of the software development processes used on your project? [Never, Rarely, Sometimes, Most of the Time, Always]

reach the target demographic (79.8% working on Scientific Computing Software).

Project Size

In Figure 2.3 we show the distribution of responses to the question about the number of FTSs currently on the project (Question GQ2). Most respondents worked on smaller teams. As smaller teams may be likely to use fewer metrics, this distribution could impact the findings of the study. Based on prior work by Carver and Heaton [10] (see Section 2.2.1), we know that many smaller teams at least unconsciously use agile software process and therefore are likely to choose the subset of software engineering practices and metrics they find useful.

Project Role

A developer's project role(s) could affect his/her perception of software metrics (Question GQ3). Identifying the distribution of respondent roles provides two types of

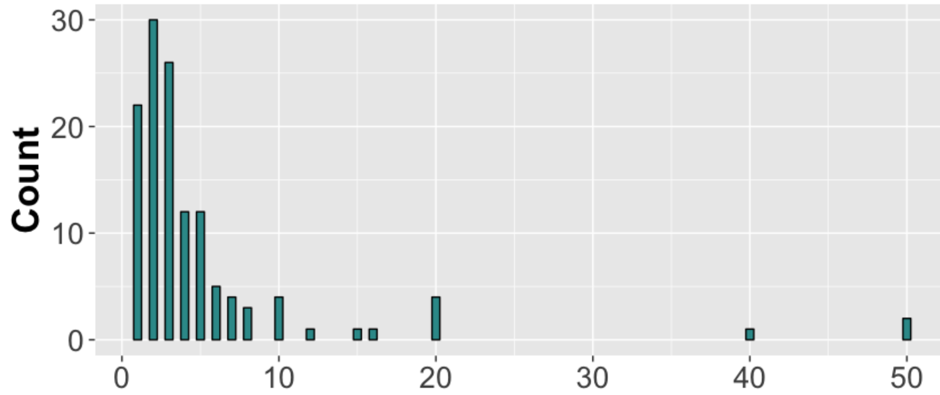

Figure 2.3: Number of Developers

insights: (1) it helps us judge whether the survey reached a broad, diverse set of research software developers, and (2) it allows us to examine whether people in different types of roles favor different types of metrics. The results in Figure 2.4 show that the respondents were skewed more towards technical roles (e.g. developers and architects) than towards non-technical roles. Note that because respondents could choose more than one role, the total in the figure is larger than the total number of respondents.

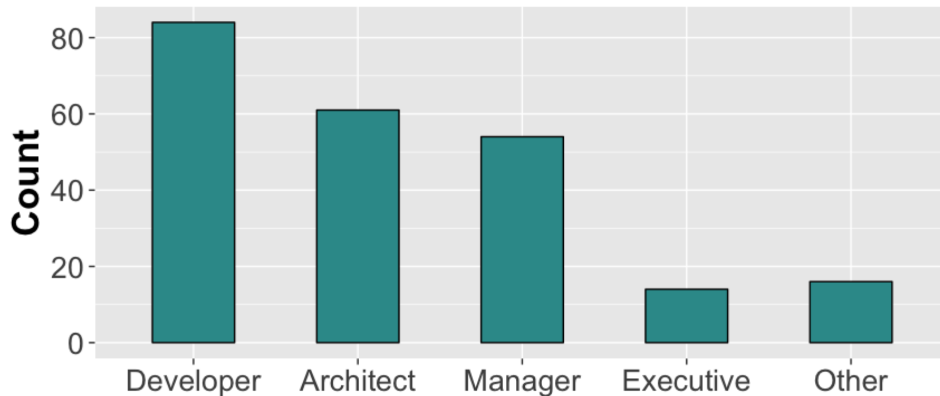

Figure 2.4: Respondents' Role on Project

Project Development Stage

Project stage helps determine which types of metrics may be most useful. Question GQ4 lists the choices for project stage. As we show in Figure 2.5, the projects represented by the survey respondents were overwhelmingly in the *released* stage. This result is important because projects at that stage should have already established metrics programs that they deem useful for monitoring development and early-stage maintenance.

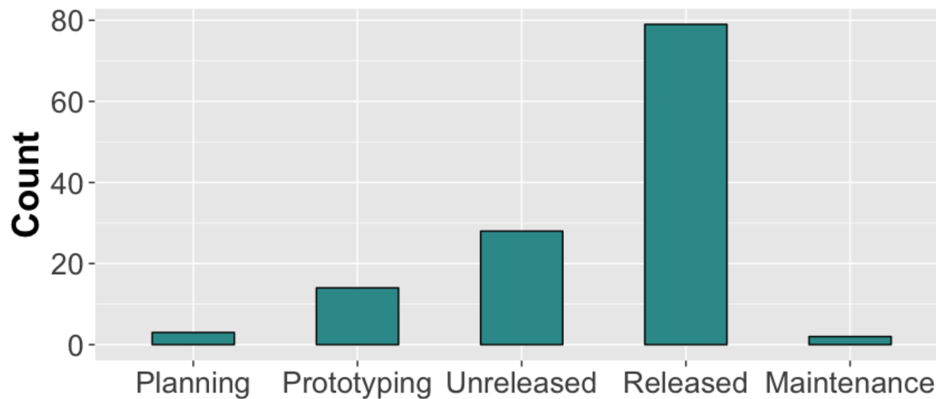

Figure 2.5: Project Stage

2.4.2 RQ1: What is the overall level of metrics knowledge and use by research software developers?

Regarding the respondents' general knowledge of metrics (Question MQ1), the majority indicated they had *low* or *very low* knowledge of metrics (Figure 2.6). Regarding the respondents' overall perception of the usefulness of metrics (Question MQ3), just under half of the respondents indicated they *never* or *rarely* found metrics useful (Figure 2.7).

However, respondents were able to name so many SE-related metrics in the free-form response questions. While the respondents reported items in the free-response questions that were not metrics by the traditional definition, they did report most of the metrics that appear in classic metrics texts [25].

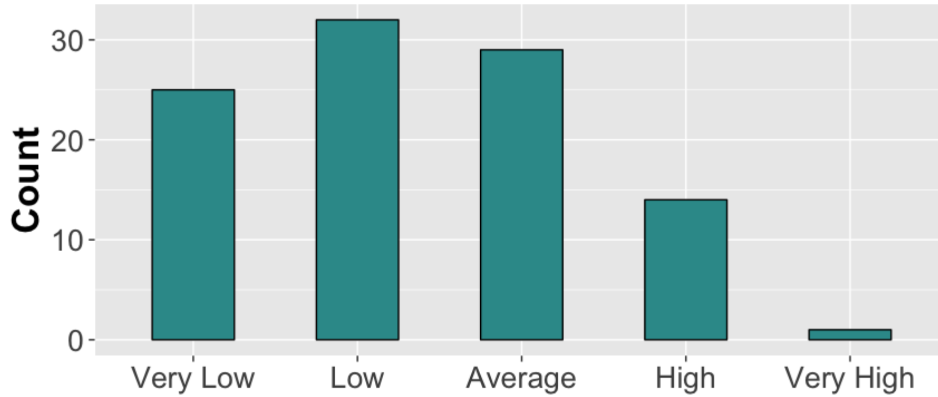

Figure 2.6: Knowledge of Metrics

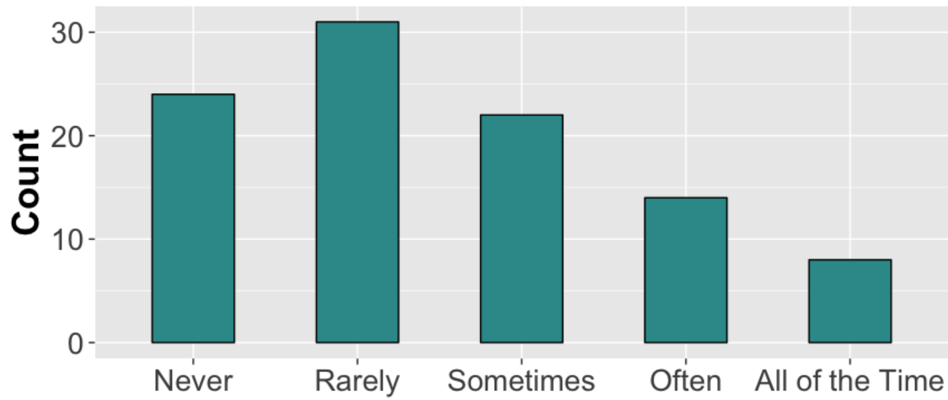

Figure 2.7: Perceived Usefulness of Metrics

2.4.3 RQ2: What is the relationship between knowledge of metrics and their perceived usefulness?

One would expect a relationship between general knowledge of metrics and perceived usefulness of those metrics. To determine whether this relationship is present in our results, Table 2.1 shows the comparison of the respondents' general knowledge of metrics (Figure 2.6) with their perception of the usefulness of those metrics (Figure 2.7). A tau-c test for independence (appropriate for comparing two ordinal variables) shows that these two distributions are not independent ($p < .01$) indicating that general knowledge of metrics and perceived usefulness are related.

Table 2.1: Perceived Usefulness of Metrics vs. Knowledge of Metrics

		Knowledge					
		Very Low	Low	Average	High	Very High	Total
Usefulness	Never	15	5	3	1	0	24
	Rarely	6	12	8	4	1	31
	Sometimes	2	6	9	4	0	21
	Often	1	7	5	1	0	14
	Always	0	1	4	3	0	8
	Total	24	31	29	13	1	98

In general, we found that as perception of the usefulness of metrics increases so does the likelihood that research software developers will use those metrics to evaluate individuals and teams. This disparity suggests that research software developers may struggle to adopt metrics in their software unless they are better informed about their merits. Based on our results, we found that smaller teams tend to have stronger negative perceptions of metrics. Among all metrics, however, both small and large teams reported the greatest knowledge of code metrics.

2.4.4 RQ3: Which metrics are most commonly used?

Next, we conducted a qualitative analysis of the specific metrics that respondents indicated they knew (MQ2) and used (MQ4) in their projects. In total, the respondents listed 89 unique metrics, indicating they were aware of a large number of metrics. We grouped these 89 unique responses into the following six high-level categories (The detailed list of metrics can be found in the paper appendix):

- **Code Metrics** – *includes those metrics that measure complexity (e.g. McCabe, # of classes, and coupling) and that measure other characteristics of code (e.g. # of clones, and defect density).*

- **Process Metrics** – *includes metrics that are collected over longer periods of time and provide insight into the software development process (e.g. productivity, cycle time, or # of commits).*

- **Testing Metrics** – *includes metrics that measure and monitor testing activities by giving insight into test progress, productivity, and quality (e.g. code coverage or # of tests).*

- **General Quality Metrics** – *includes metrics related to desirable properties of software that are not easy to measure as part of the development process or through analysis of the source code (e.g. interoperability, portability, or sustainability).*

- **Performance Metrics** – *commonly of interest for software executing on high-performance computing platforms, the metrics address execution time, storage (e.g. RAM or disk space), or scalability (e.g. time vs. CPUs/cores).*

- **Recognition Metrics** – *includes metrics that measure how a project or its developers quantify outside interest in their work (e.g. citations or downloads).*

It is interesting to note that in addition to the four categories that are commonly found in the software metrics literature (Code, General Quality, Process, and Testing), we identified two categories of metrics (Performance and Recognition) that are not found in the traditional software metrics literature. These new categories are often of interest to research software developers working in high-performance computing environments. Recognition is particularly timely as research software developers are increasingly interested in being recognized and receiving proper credit for developing research artifacts such as software, tools, and libraries [27, 41–43]. Table 2.2 provides an overview of the responses.

Respondents reported *performance* and *testing* as the most known and used metrics. The presence of performance metrics is reasonable given that many research teams use high-performance architectures and computers to do their work. The high use of testing metrics is a positive result because our experience suggests that mean teams lack resources for performing adequate testing.

One interesting result relates to *code* metrics. The results in Table 2.1 show that respondents reported the most unique code metrics and reported the highest frequency of

Table 2.2: Categories of Metrics Used

Category	**Number of Unique Metrics**	**Known (frequency)**	**Used (frequency)**
Code	24	94	17
General quality	14	23	16
Performance	13	41	33
Process	21	28	9
Recognition	5	15	8
Testing	12	48	24

knowing code metrics across all six categories. Conversely, the reported use of code metrics was dreadfully low compared with the ratio between known and used for the other categories.

The data in the survey did not provide the type of information necessary to explain why this result may have occurred. Nevertheless, it is both interesting and potentially worrisome. One potential explanation is that while respondents were aware of many different code metrics, they did not believe that these metrics were actually useful in their research software projects. Further research is needed to better understand this discrepancy and identify and necessary solutions that can reduce the gap.

Based on the results, we make some additional observations about the metric categories:

- *Testing metrics* – Respondents used testing metrics second only to performance metrics. This result is encouraging, considering their appearance in the SE literature [25] and TDD [56].

- *General quality* – While these metrics do not always correspond directly to methods established in SE literature, they are interesting because they shed light into how research software developers view quality in general. We were also encouraged to see interest in sustainability, which is an area of growing importance within the research software community [41–43].

- *Performance metrics* – These metrics are clearly of value on the types of systems typically used by research software developers. When the software is written to run on a high-performance computer, for example, lack of performance is a negative characteristic.

- *Process metrics* – Respondents reported high usage of metrics of interest to agile software developers. Given that many of the responses came from small-to-medium sized teams, most of these suggest the use of agile processes.

- *Recognition* – From a traditional SE perspective, this set of metrics would be somewhat unexpected. Respondents reported many metrics as being significant for addressing recognition. The presence of these metrics reinforces the current notion that developers of research software need more and better ways to formally track and quantify their contributions to research.

Finally, in Figure 2.8, we show the results of survey questions MQ5 and MQ6 asking whether research software teams use metrics, for individual or team evaluation. As the figure shows, the vast majority of the respondents indicated that metrics were *never* or *rarely* used to evaluate individual or team productivity.

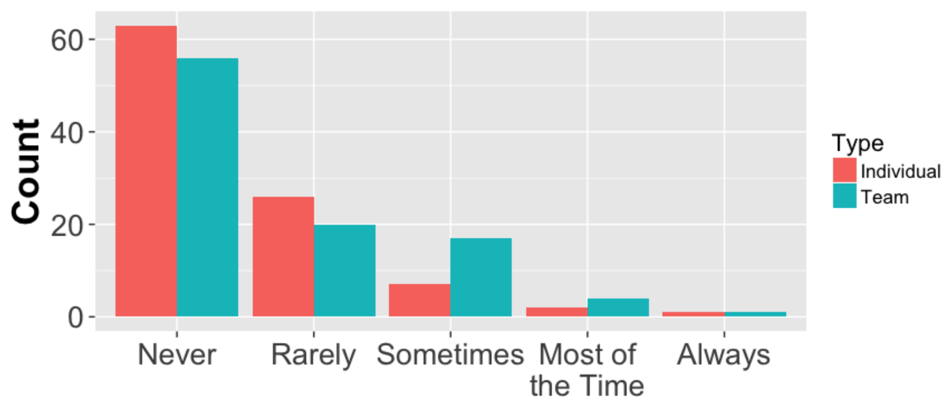

Figure 2.8: Frequency of using Metrics for Evaluation

Similar to the analysis above, we analyzed whether there was a relationship between

perceived usefulness of metrics (Figure 2.7) and the use of metrics for evaluation (Figure 2.8). For both individual and team productivity, the tau-c test for independence showed the distributions were not independent ($p < .01$). Once again, this result shows a relationship between perceived usefulness of metrics and the likelihood of using those metrics for evaluation.

2.4.5 RQ4: To what extent do the demographics of the respondents affect their perception and use of software metrics?

In this section, we examine whether the demographics defined in Section 2.4.1 affect overall knowledge of metrics, overall perceived usefulness of metrics, knowledge of specific types of metrics, or use of specific types of metrics. For each demographic, we describe the analysis separately in the following subsections.

To facilitate the analysis (and appropriately use the number of data points we have), we divide the values for each demographic into two categories, as defined below. These divisions do not result in equal sized groups, so in the following analysis we normalize the data. First, for the influence of the demographics on overall knowledge and overall perception of usefulness, each respondent could give only one answer, so we analyze the responses as percentages (e.g. the percentage of respondents in the group that gave each answer). Second, for the influence of the demographics on the knowledge and use of specific metrics, each respondent could give multiple answers, so we normalize the responses with the size of the group and report the number of each type of metric per respondent (e.g. how many code metrics were reported per person in the group).

Influence of Project Size

We grouped respondents into: *small teams* (less than five participants) and *large teams* (five or more participants). The analysis shows that respondents from smaller projects appear to have less overall knowledge of metrics (Figure 2.9) and see metrics as less useful (Figure 2.10) than respondents from larger teams. Both of these results are

significant using a χ^2 test with p-value $< .001$. This result could be due to the typically smaller amount of resources smaller teams have to devote to metrics. Note that in both cases the responses are generally skewed leftward, which is not surprising given the results in Figures 2.6 and 2.7.

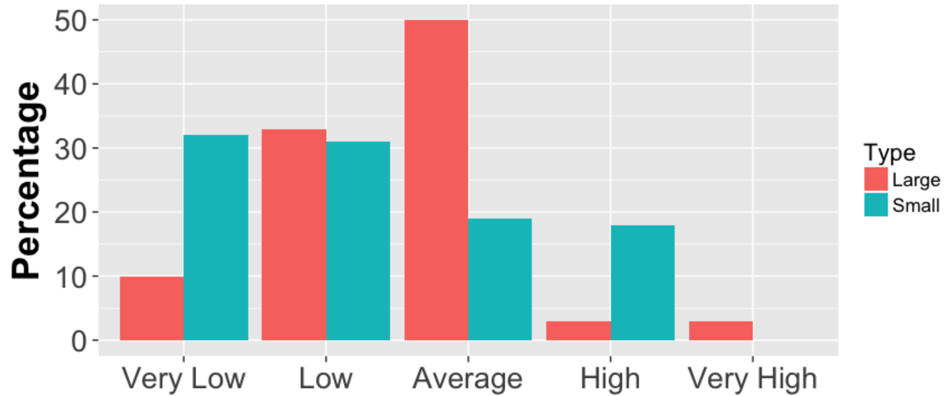

Figure 2.9: Influence of project size on overall knowledge of metrics

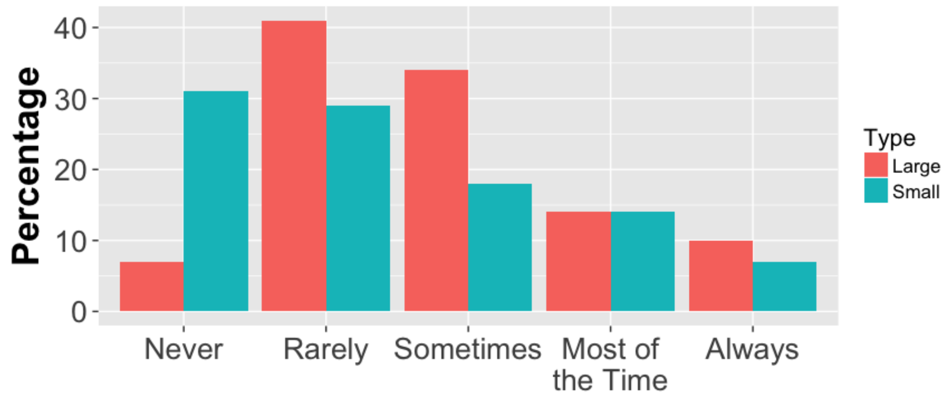

Figure 2.10: Influence of project size on perceived usefulness of metrics

Turning to specific metrics, larger teams have knowledge of more metrics in four of the six categories (Figure 2.11). Conversely, use of metrics is more consistent between large and small teams with two notable exceptions. Large teams use about 2.5 times as many *code* metrics as small teams and small teams use about three times as many *recognition* metrics as large teams.

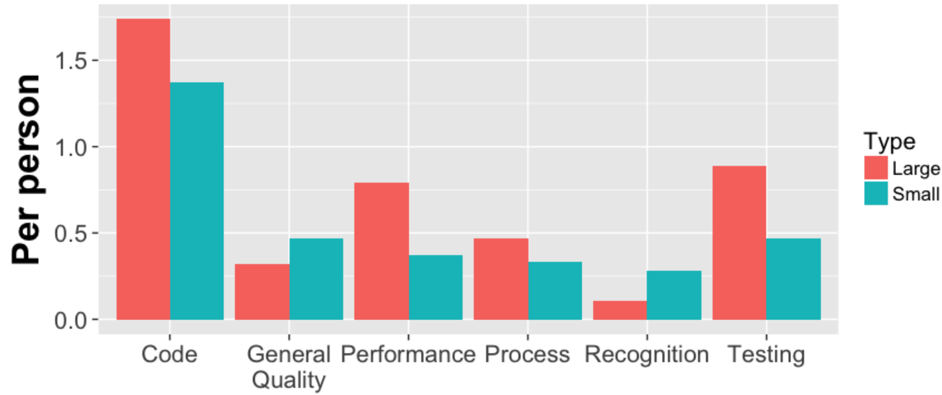

Figure 2.11: Influence of project size on known metrics

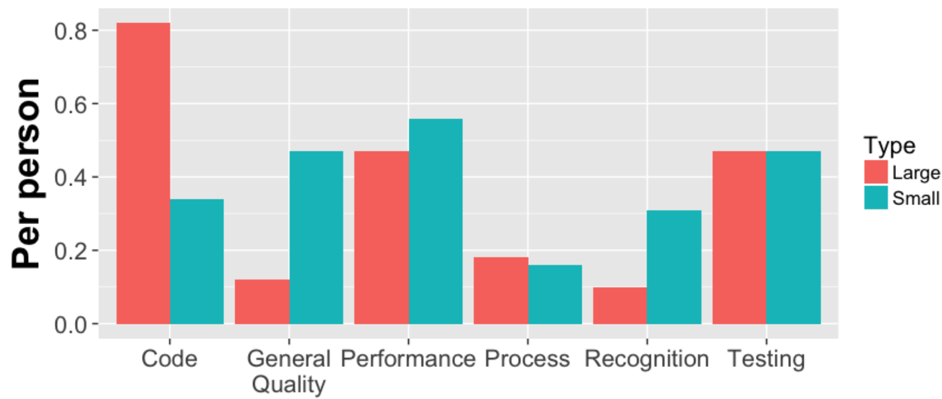

Figure 2.12: Influence of project size on used metrics

Influence of Project Role

We grouped respondents into: *technical* (consisting of developers and architects) and *non-technical* (consisting of manager, executives, and other). Note for this analysis, respondents can appear in both categories if they gave both types of responses (see Section 2.4.1), resulting in total percentages greater than 100. The analysis did not show any significant effect of project role on either overall knowledge or overall perception of usefulness. The fact that we allowed respondents to choose multiple roles in response to GQ3 resulted in respondents who were in both categories. While, given the nature of research software development, this result is not surprising, it likely contributed to the lack

of significant findings.

Influence of Project Stage

We grouped respondents into: *released software* (including those in released and maintenance phases) and *unreleased software*. The analysis showed that level of overall knowledge (Figure 2.13) and overall perception of usefulness (Figure 2.14) differed significantly between the groups (p-value $< .01$ on the χ^2 test in both cases). Examining the distributions, we can observe that respondents with unreleased software had more people with *very low* and more people with *high* knowledge than those with released software. The results for perceived usefulness mirror these results. This result could be caused by respondents at different phases of unreleased software viewing metrics differently.

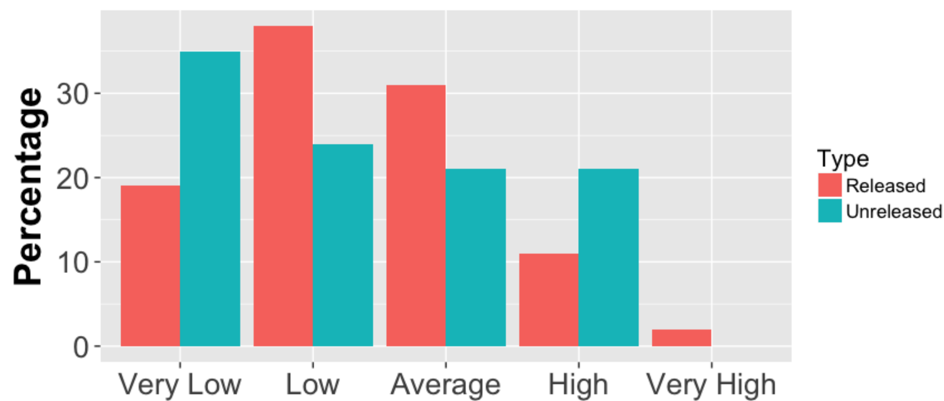

Figure 2.13: Influence of project stage on overall knowledge

For knowledge of specific types of metrics, project stage had very little differentiating effect. Conversely, for use of specific types of metrics (Figure 2.15), respondents from unreleased software found *performance* metrics more useful while respondents from released software found *testing* metrics more useful.

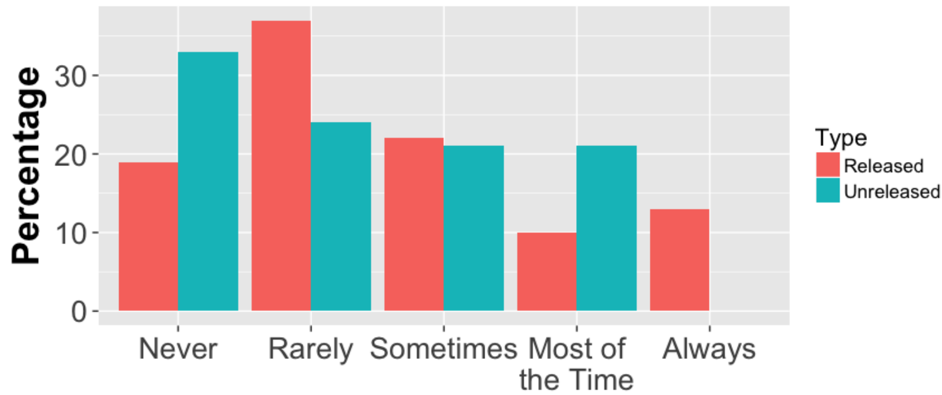

Figure 2.14: Influence of project stage on perceived usefulness

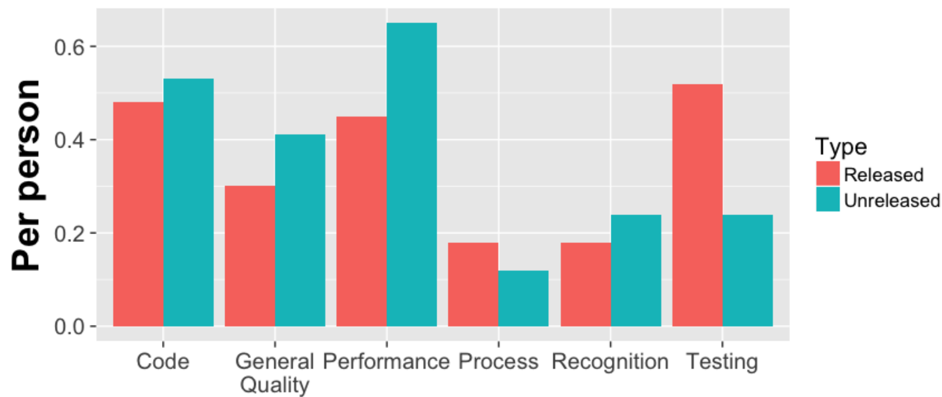

Figure 2.15: Influence of project stage on specific metrics used

2.4.6 RQ5: Do research software developers perceive code complexity as a problem?

The survey respondents perceived code complexity to be a problem. In Figure 2.16, we show the responses to survey question CQ1. Most respondents indicated code complexity is a problem at least *sometimes*. Interestingly, while respondents considered complexity to be a problem, the respondents answers to survey question CQ2 (Figure 2.17), the vast majority said they *never* used code complexity metrics, with only a small number using them more often than *rarely*.

While this result was not entirely a surprise, given our experiences of observing

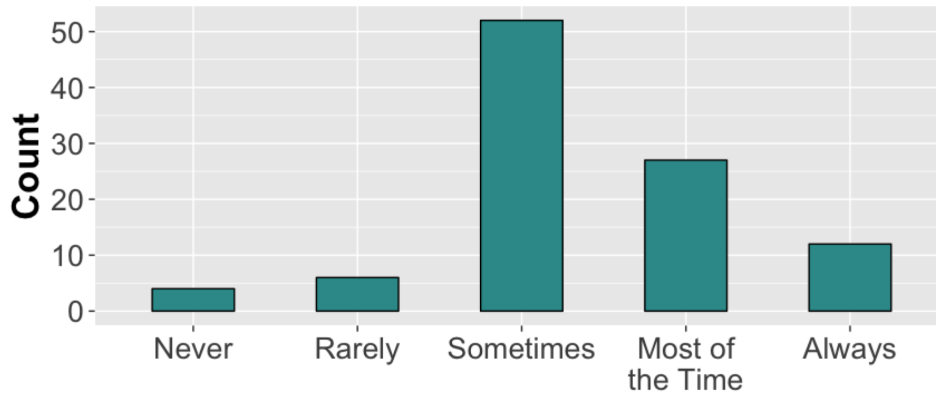

Figure 2.16: Frequency of Code Complexity Problems

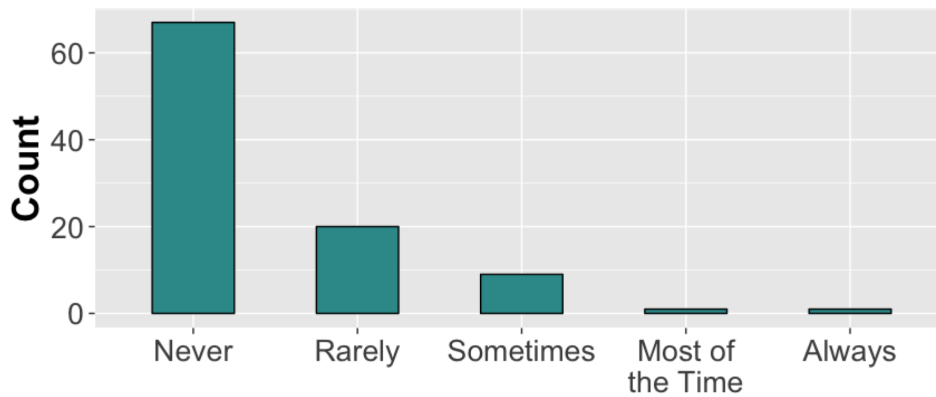

Figure 2.17: Use of Code Complexity Metrics

complexity in research software, it was good to see respondents self-reporting that this issue is worthy of attention. Furthermore, this result suggests that the research community has significant interest in code complexity but struggle to adopt relevant metrics in their software projects.

2.4.7 RQ6: Is the frequency of complexity problems related to the use of or perceived helpfulness of metrics?

Of those that used complexity metrics (e.g. rarely or above in Figure 2.17), the responses to CQ3 indicated that the majority found the metrics were *never* useful (Figure 2.18). Finally, there was no significant relationship between the frequency of complexity

problems and the use or helpfulness of complexity metrics.

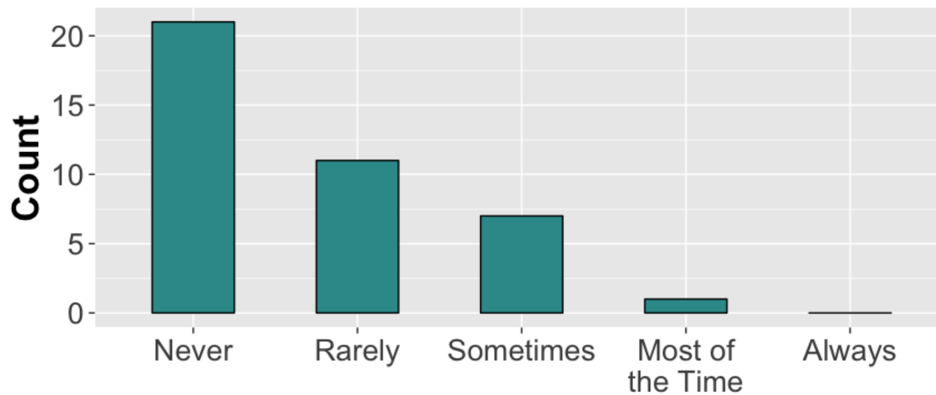

Figure 2.18: Usefulness of Code Complexity Metrics

This result is surprising (and somewhat troubling), given the prevalence of code complexity problems. We need further study to understand whether the low use of complexity metrics is caused by their perceived lack of relevance or by the lack of support for those metrics in the programming languages research software developers commonly use, or by some other reason.

2.4.8 RQ7: To what extent do research software developers follow software process?

To obtain an overall understanding of whether research software developers follow process, the survey asked two questions. First, survey question PQ1 focused on the frequency with which the respondents' teams followed software process. Second, survey question PQ2 focused on the frequency with which the respondents individually followed software process. Figure 2.19 summarizes and compares the results from these two questions.

Based on the results, we can make the following observations. First, the majority of the respondents indicated that both their teams and themselves individually follow a defined software development process at least *sometimes*, with a large group following process *most of the time* or *always*. It is encouraging to see this relatively high level of

process use for both teams and individuals.

Second, examining the overall trend between individuals and teams, we observe that use of process by the respondents' teams and by the respondents individually is consistent at both ends of the spectrum (*never* and *most of the time/always*). The primary differences occur towards the lower middle (*rarely* and *sometimes*, which total about the same when considered as a group). As these scales are ordinal, we performed a Mann-Whitney test to determine whether there is a difference in the responses between these two responses. Not surprisingly based on the figure, the differences are not significant (U = 4403, p = 0.6282).

The lack of a significant difference between teams and individuals allows us to make some general observations. If a team uses process, the individual is likely to use process. If a team does not use process, the individual is not likely to use process. Given that our responses come primarily from smaller teams, this observation suggests that individuals can have an important effect on what a team does when it comes to process (positive or negative). As we did not seek to determine the influence of individual over team (or vice versa), further research may be needed to understand the dynamics of research software teams.

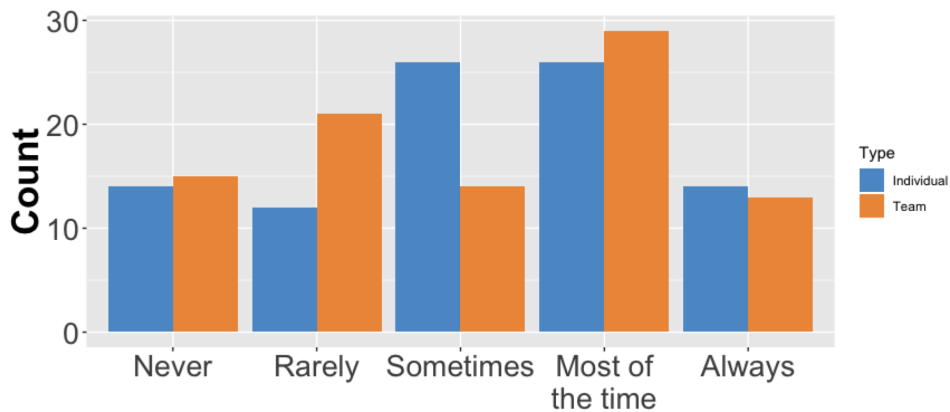

Figure 2.19: Frequency of following process

Overall, the above findings are encouraging, because if a research team follows a defined process, there is increased potential that team could use process-related metrics as

well as other best practices for software development, assuming they collect proper data at all phases of software development.

2.4.9 RQ8: What value do research software developers place on software process?

Survey question PQ3 asks the respondents to indicate the value they see in following a defined process. According to Figure 2.20, with a few exceptions, most at least see *moderate* value in following a software development process with a relatively large number finding *high* and *very high* value. This result suggests that many research software developers may be interested in following a process for which they see value. The result confirms what we have learned from the literature review, where there are mixed results on which particular software development process research software teams follow, but the common element is that development process is of value to research software teams. While we did not ask respondents to explain why they did or did not values software process, we will explore this question in future studies.

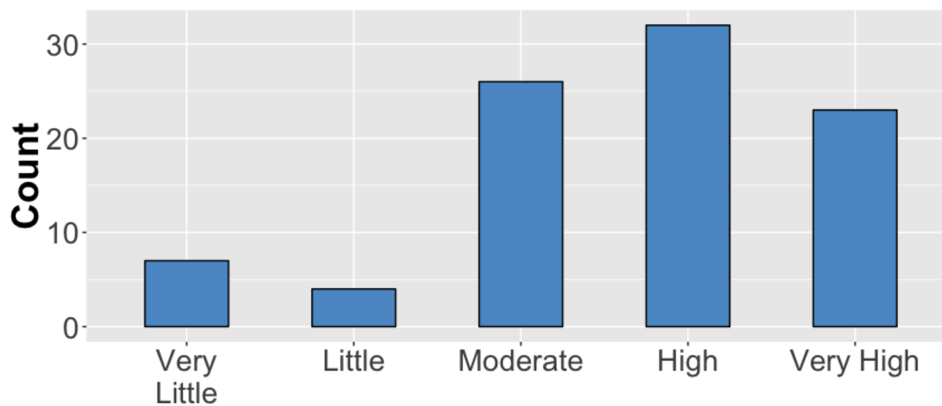

Figure 2.20: Frequency of value seen in process use

2.4.10 RQ9: Is there any relationship between the valued placed on software process (RQ8) and the use of software process (RQ7)?

It would be reasonable to expect that someone who values software process would be likely to use software process. To check whether this assumption holds, we compared the

responses to PQ2 and PQ3. We focused on PQ2 (individually following process) because the respondent has more control over whether he or she follows process than they do over whether the team follows process. In Table 2.3 we show the distribution of responses between these two variables. There is a significant (χ^2 (16, N=92) = 92.56, p < .01) relationship between a respondent's perceived value of following a defined software development process and their likelihood of individually following a defined process. This result suggests that one way to increase the level of software process in research software is to help research software developers understand the value of following software process.

Table 2.3: Perceived Value of Process vs. Use of Process

Follow \ Value	Very Little	Little	Moderate	High	Very High	Total
Never	6	1	4	3	0	14
Rarely	0	1	9	2	0	12
Sometimes	1	1	11	13	0	26
Most of the Time	0	1	1	13	11	26
Always	0	0	1	1	12	14
Total	7	4	26	32	23	92

2.4.11 RQ10: Which software development processes do research software teams use?

Given the positive perception about following a defined software development process, it is interesting to see which specific process the respondents used. Survey question P4 asked respondents how often the followed a set of popular software process models. The results in Figure 2.21 show research software developers prefer *Agile* and *Ad Hoc* processes. While many of the respondents used *Use-Case Methodology* and *Rapid Application Development* at least sometimes, most respondents never or rarely used *Waterfall Model* and *Spiral Model*.

It is interesting to note the apparent discrepancy between the relatively high frequency with which respondents follow process (Figure 2.19) and the relatively low frequency with which the respondents follow any of these process models. It is likely that responds either

follow different models or were not aware of the formal names of the processes they follow. In fact, our literature review pointed to the belief that agile-like approaches (not necessarily any of the named Agile approaches) may be a better fit for research software and these results support that belief.

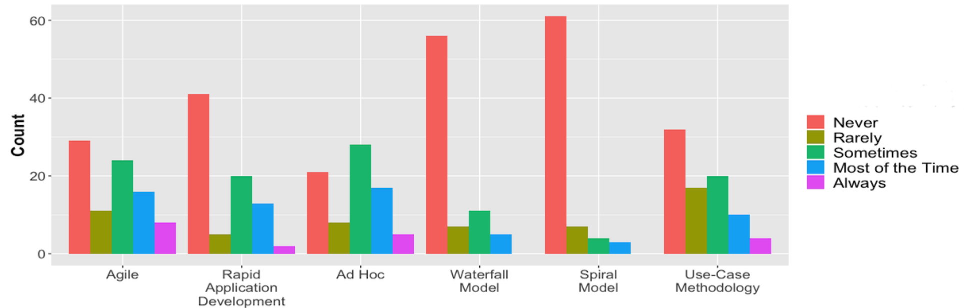

Figure 2.21: Frequency of using particular process

2.4.12 RQ11: To what extent do research software teams use software process metrics?

Survey question PQ5 asked the respondents about the use of process metrics to evaluate the effectiveness of the software development processes in their teams. Consistent with the results reported in our other paper [22] the use of process metrics for evaluation of effectiveness (Figure 2.22) is quite low. This result is surprising because, without evaluation for effectiveness, teams could be using inappropriate processes. Therefore, one area of potential improvement for research software teams is to employ metrics for evaluating the processes they are using to help determine whether any changes are necessary.

2.4.13 RQ12: To what extent do the demographics of the respondents affect their perception and use of software process?

Using two of the four demographics defined earlier, we analyze whether they impact how respondents' answers about following and valuing process. For each demographic, we split the sample into two categories. We used logic in determining the split points, rather than trying to ensure equal sized groups which were not meaningful. Therefore, because the groups are not of equal size, we normalized the data and analyze the responses as

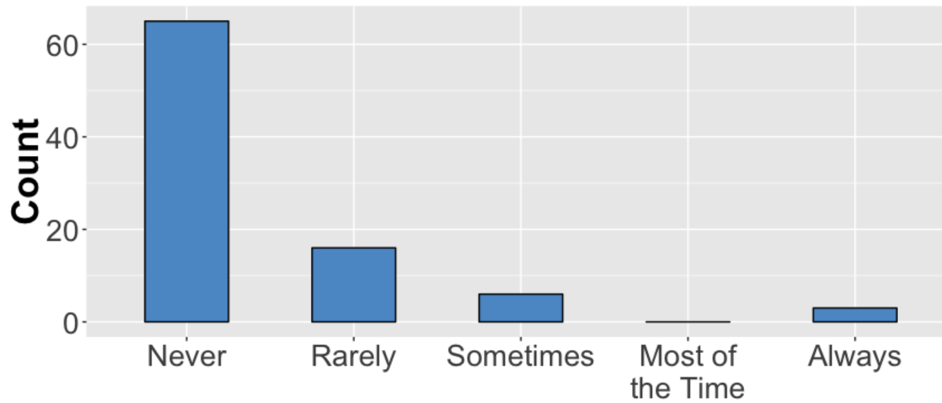

Figure 2.22: Frequency of process metrics use

percentages. We describe the analysis separately for each demographic.

Influence of Project Size

We divide the respondents into small teams (less than 5 people) and large teams (5 or more people). From documentation on agile models (e.g. the Scrum Guide [29]) we know that agile teams, ideally, are not so large (between 3 and 9 members). We analyzed the data separately for teams (Figure 2.23) and individuals (Figure 2.24). In both cases the differences between large and small teams are significant (Mann-Whitney $p < .001$).

In general, it is not surprising that large teams followed a defined process more often than small teams (Figure 2.23). From the literature (and anecdotally) we know that larger teams are more likely to embrace traditional software process.

The reason may be the small number of resources smaller teams get to invest in their software projects. When we look at the data about the effect of team size on whether individuals follow process (Figure 2.24), the results look a little different. Individuals on small teams appear to follow process more frequently than small teams as a whole.

In terms of value seen in following a defined software development process, Figure 2.25 shows that large teams tend to see more value in following process than small teams. This result is significant ($U = 1951$, $p < .001$). Again, this result makes sense because the larger

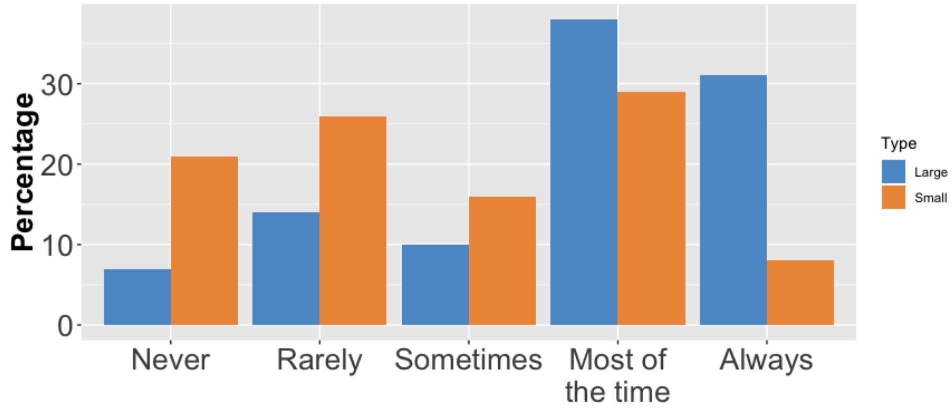

Figure 2.23: Influence of project size on process following as a team

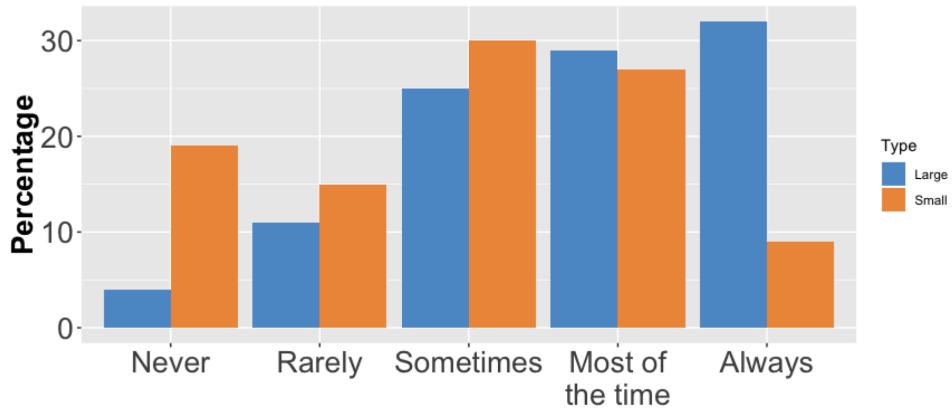

Figure 2.24: Influence of project size on personally following process

a team gets, there is more need for a defined process to manage the development.

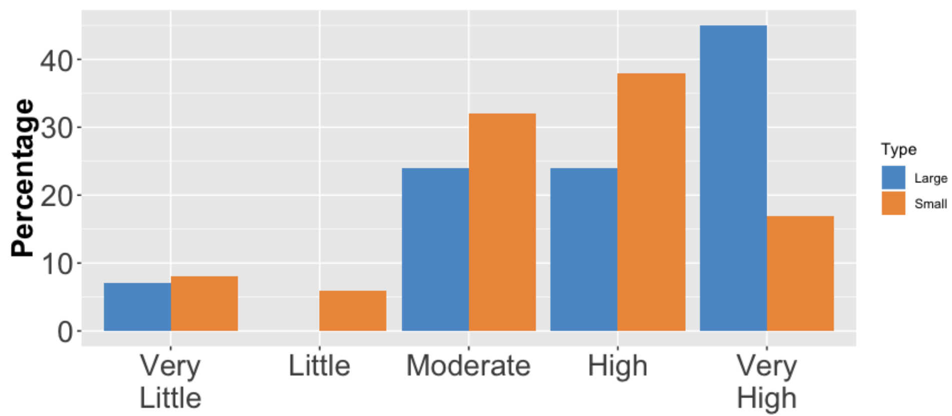

Figure 2.25: Influence of project size on value seen

Influence of Project Stage

We split the respondents based on whether their project was Released (including released and maintenance phases) or Unreleased (Including planning, prototyping, and Unreleased phases).

In terms of following defined processes, the teams of the respondents working on released software followed defined process more than the teams of those working on unreleased software (Figure 2.26). This difference is significant based on the Mann-Whitney test ($U = 5995$, $p < .0125$). The results for respondents individually following process mirror the results for the teams.

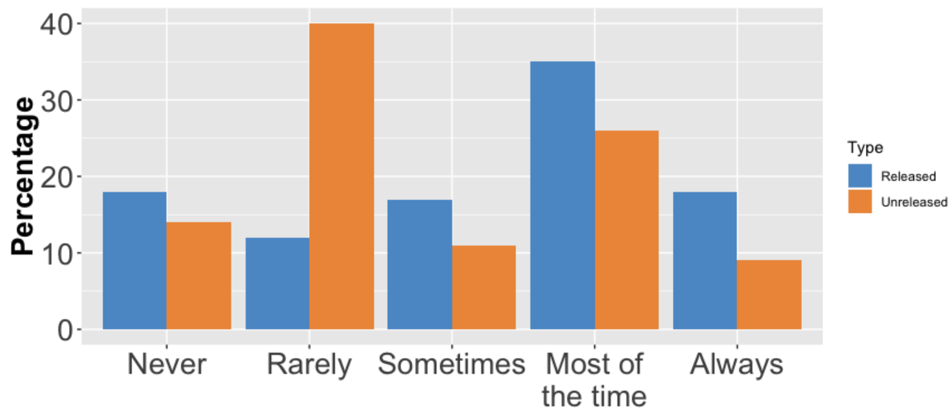

Figure 2.26: Influence of project stage on process following

In terms of valuing process, the results in Figure 2.27 show that respondents working on released software value process more than those working on unreleased software. The difference is significant based on the Mann-Whitney test ($U = 6135$, $p < .001$). Again, this result is not surprising as those who have released software have had the chance to see the value that following software process can have.

It is also not surprising that people from both the stages tend to not use process metrics to evaluate the effectiveness of the software development process.

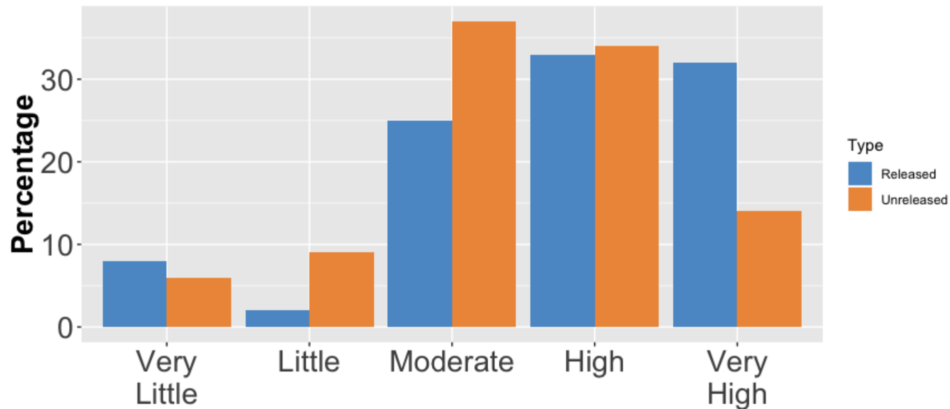

Figure 2.27: Influence of project stage on value seen

2.5 Threats

In this section we discuss the threats to validity from the survey.

2.5.1 Internal Threats

This survey faces two primary internal validity threats. The first is the potential for introducing bias through the survey design. Because the members of the target survey population are not traditional software developers, it is possible that they lacked the necessary knowledge to properly answer the survey. To prevent introducing bias in this situation, we purposefully phrased survey questions in a neutral manner (without providing the names or types of any metrics), thereby allowing the respondents to reveal their own understanding of metrics.

The second potential validity threat is selection. It is possible that some survey respondents were not actually research software developers. Although the vast majority of respondents indicated that they are working on research software, some did not. Given the nature of the research software domain, it is possible that some of the survey respondents work on software that supports research software (like middleware or tools) rather than directly on research software itself. Nevertheless, the number of responses (129) represents a relatively large set of responses for a community that is likely smaller than other

communities traditionally surveyed in software engineering research. Therefore, we find this threat to be minimal.

2.5.2 External Threats

The survey sample may not be representative of all research software developers. Although we took great care to send our survey to research software developers, due to the particular mailing lists we used, it is possible that some segments of this population, like those from US-based national labs and HPC-related groups are over-represented in the sample. These segments of the population are clearly research software developers, but they may not represent the way all research software developers think.

Furthermore, the respondents who chose to respond to the survey may not be an accurate representation of all research software development groups. For example, members of corporate research software development groups may be even more inclined to embrace more formal/defined software processes. Conversely, smaller research software teams with less formal support and resources may be less likely to use well-defined software processes and the related metrics.

2.5.3 Construct Threats

It is always possible that survey respondents misunderstand the survey questions. In our case, however, we went out of our way not only to provide questions but to give clear directions for how to respond to those questions, without biasing the respondents.

The other primary construct validity threat is whether the respondents understood the software engineering and software metrics concepts in the same way as we intended them. While we did not specifically evaluate this issue in the survey, previous surveys have shown that research software developers generally understand SE concepts in the traditional manner. Furthermore, based on the fact that respondents reported many of the metrics traditionally included in the SE literature as well as some that are specifically important in the research software domain, we are reasonably confident that the questions were clear.

2.6 Conclusion

In this chapter, we report on the results of a survey of research software developers to assess knowledge and use of software metrics. In addition to specific metrics, the survey also sought to understand whether research software developers found software development processes to be useful. In all the respondents, most of whom were true research software developers completed the survey. The results showed that while research developers knew and used metrics (in general), they did not as commonly use traditional SE metrics in actual projects. Although research teams report code complexity to be a nagging problem, the research software developers who responded to the survey only used code metrics on a limited basis on their projects and generally do not perceive them positively. Conversely, the survey respondents appeared to be relatively familiar with code complexity metrics, based upon the fact that this category of metrics represents the largest set of responses to the free form questions about metrics.

Furthermore, the results show that research software developers are very familiar with and frequently use performance and testing metrics. The use of performance metrics is logical given that many research software developers develop mathematical or scientific algorithms that are expected to have good performance or else risk not being used in real-world applications.

When it comes to software development process, most respondents reported that they follow a defined software development process and find it valuable. Not only do respondents find software development process to be of value, they actually prefer using a defined software development process. Although the vast majority of responses suggest the use of agile and ad hoc processes, traditional software development processes do see actual use in established research software projects.

An encouraging result is the identified relationship between a respondent's perceived value of using a defined software process and their likelihood of actually using a defined software process. An interesting finding is that the use of agile methods includes the use of

lighter weight methodologies, including Use-Case Methodology and Rapid Application Development in their work.

A somewhat discouraging finding is that although respondents see value in process, they rarely use process metrics to evaluate the success of software process. In the absence of process metrics, it is difficult to imagine how quality control can be ensured, because process effectiveness is typically measured at every stage to reduce defects in production.

In conclusion, this work shows that various software metrics and process could be of value to research software development teams. While work remains to be done to increase knowledge of metrics within this community, we hope that this work can be a first step toward helping research development teams see the potential merits of using metrics and process that are based upon the SE methods that they already employ in their projects.

CHAPTER 3

PEER CODE REVIEW IN RESEARCH SOFTWARE

3.1 Introduction

Researchers in a number of scientific, engineering, business, and humanities domains increasingly develop and/or use software to conduct or support their research. We refer to this software collectively as *research software*. Without such research software, it would be difficult or impossible for many researchers to do their work [55]. Research software includes both software for end-user researchers, such as software for weather forecasting or molecular dynamics simulation, and software that provides infrastructure support, including messaging middleware, scheduling software, and various mathematical, scientific, or statistical libraries. Results produced by research software help researchers predict natural phenomena and make decisions to support critical needs. Researchers need high quality software that will produce trustworthy results and function properly in mission-critical situations. Therefore, researchers need to follow appropriate software engineering practices to help ensure the quality of the software.

Historically, developers of business/IT software have employed various types of testing techniques to improve the quality of their software. However, developers of research software have found it more difficult to employ some of the traditional software testing techniques [39]. The complexity of the underlying research domains leads to research software that has complex computational behavior. This complexity, along with the fact that the expected outputs are often unknown, make it difficult to define appropriate tests

and identify input domain boundaries [33]. In many cases, the input space of research software is so vast that it is not feasible, or even possible, for a developer to create a test suite that adequately exercises the limits of the software. Therefore, while testing is useful, it is generally not sufficient to ensure the quality of research software.

Conversely, peer code review is a lightweight, asynchronous method for ensuring high-quality code [1]. Peer code review is a systematic examination of source code by peers of the software's developer to identify problems the developer can then address. Recently, commercial organizations and open source projects have been adopting peer code review as a more efficient, lightweight version of the older, more formal inspection process [63]. While peer code review is effective and prevalent in open-source and commercial software projects, it remains underutilized in research software.

By employing peer code review in their projects, research software developers will see benefits both in the short term, through higher quality scientific results produced by high-quality software, and in the long term, through creation of more maintainable software. The higher quality scientific results occur because developers focus their attention on the code itself to identify mistakes, inefficiencies, and other aspects of the code that need improvement. The improved maintainability arises because as team members start reviewing each others's code, they begin writing more readable code to enable the peer-review process. Code that is more readable and easier to understand is also more maintainable over time.

In addition to improving general software quality, the use of peer code review has other specific benefits in the research software domain. Unlike traditional commercial/IT software, research software developers are often exploring new scientific or engineering results, which may be unknown *a priori*. The lack of an oracle makes it difficult for developers to create adequate tests that can diagnose whether a result is a new insight from a simulation or is the consequence of a fault in the software. Even in cases where the expected output is known, the complexity of the software often makes it impossible to

adequately test all important configurations of the software and input data. Conversely, when a person conducts a code review, he or she is able to analyze the underlying algorithm and identify problematic conditions. Therefore, while peer code review is essential for any type of software, it is even more important for research software.

However, despite the benefits peer code review can provide, developers of research software do not yet apply it widely. In addition, the literature does not contain any studies on the use of peer code review in research software. Therefore, to better understand how to increase the usage of peer code review in research software development, this paper focuses on characterizing how research software developers perform code review. The primary goal of this study is to *better understand the practices, difficulties, impacts, and potential areas of improvement of code review in research software development.* To gather this insight, we interviewed and surveyed research software developers from a wide array of projects about their current practice and the challenges they face.

The key contributions of this chapter are:

- The first study of peer code review use in research software.
- An overview of the current code review practices in research software development.
- Identification of the types of defects identified during the code review process.
- Positive and negative experiences research software developers have regarding code review.
- Impacts of the the code review process on research software projects.
- Challenges and barriers developers face during code review.
- Potential areas of improvement in the review process.

The remainder of this paper is organized as follows. In Section 3.2 we provide the necessary background information to motivate the research questions explored in this

study. In Section 3.3 we discuss the research questions that drive this study. In Section 3.4 we explain the study design, data collection, and data analysis procedures. In Section 3.5 we provide the detailed results. In Section 3.6 we discuss the key finding from study. In Section 3.7 we discuss the validity threats. In Section 3.8 we draw conclusions.

3.2 Background and Motivation

Because code review has not yet been studied in the context of research software, this section focuses on prior research about code review in general. Developers of both open-source and commercial software have increasingly employed peer code review as part of their development process.

Google began using code review early in its history and evolved the practice over the years. It is an important aspect of the development workflow and has become a core part of Google culture. Because code acts as a teacher to future developers, Google forces people to write code that is understandable to other developers. Google developers expect four key themes from code review: education, maintaining norms, gatekeeping, and accident prevention. Google does code review not only to solve problems but to ensure readability and maintainability [61, 66].

Open source software projects have also found peer code review a valuable method to improve software quality. Components that have higher review coverage and higher rates of participation in code review are likely to have fewer post-release defects [52]. Additionally, 75% of the changes resulting from code review relate to extra-functional improvements, i.e. code quality and documentation. Only 25% relate to functionality [4, 6, 48]. That observation indicates that code review is valuable for short term defect detection but more valuable in long term maintainability. Overall, poorly reviewed code has a negative impact on software quality in open source projects [52].

In a study comparing code review practices at Microsoft and open source projects, researches studied the effects of code review on developers perception, collaboration, and

other non-technical aspects of code review. When it comes to accepting a review request, reviewers consider their relationship with the author, the reputation of the author, their own area(s) of expertise, and the anticipated time/effort required to conduct the review. After accepting a code review request, reviewers typically start with the most familiar portions of the code before reviewing the unfamiliar parts, in which they often learn something. Besides learning, code reviews ensure that at least one or two people other than the developer are aware of code changes, thus creating project awareness among the team members. Furthermore, code reviews encourage community building and collaboration by fostering direct interactions between developers and reviewers [8].

Open source contributors come from diverse backgrounds, expertise levels, and locations. Because the quality of their contributed code varies greatly, the code reviews in open source projects emphasize maintainability and consistency. Conversely, developers from Microsoft have less variation in their code. Therefore, the code reviews can focus more on defect detection. Microsoft developers consider code review as a great source of knowledge sharing. On average a Microsoft developer spends 15%-25% of his time in code review while an open source developer spends even more time on review [52].

Code reviews should occur early and often to help developers find problems before they become more expensive to fix. It is easier for reviewers to focus on small, independent, and complete contributions. But breaking up a large change or segment of code to make small pieces may cause new problems unless a divide-and-conquer is applied [64]. Google keeps their code review process lightweight and flexible. Over time, Google has converted their process into one with markedly smaller changes and quicker reviews [66].

Peer code review also helps developers form impressions of their teammates. The quality of the submitted code is key in this impression formation process. For instance, a developer who submits simple, understandable, and self-documenting code that requires minimum review time may gain improved social status. Participation in the code review process has strong positive effects on future collaborations. As a code author's reputation

increases based on high-quality code submissions, it helps her future code submissions. In addition, as an author's reputation increases, people begin to believe more in her expertise and request her to review their code. A reviewer's perception of the expertise of the code author also influences the level of scrutiny she uses during the code review [7, 8].

Developers and reviewers also face challenges in performing code review. Reviewers find that understanding the code and the motivation for a change are the most difficult aspects of code review. In addition, many people think code review is time-consuming, with the task of understanding the change being the most time-consuming [73]. Despite being a time consuming and challenging process, developers want to participate in code review because it helps with consistency, reliability, and maintainability of the project [4]. Google, Microsoft, Facebook, and many other organizations have improved their code review process through building tools, overcoming social challenges, and investing in the process [9, 66]

3.3 Research Questions

To drive the overall study design, we identified four research questions. In this section, we introduce each research question by providing a brief motivation for why the question is important and relevant. We also describe the type of information necessary to answer the question.

RQ1: How do research software developers perform code review?

While there is extensive research on code review in commercial and open source software, there is no empirical research on the use of code review in research software. Therefore, we must first gain a basic understanding of the process research software developers follow when performing code review. To answer this question, we need to gather details about the code review process research software developers follow along with their positive and negative experiences.

RQ2: What effect does code review have on research software?

The goal of this question is to understand whether the benefits of code review seen in open source and commercial software (discussed in the previous section) also appear in research software. To provide concrete evidence about their effects, we need to gather information about how code review impacts research software.

RQ3: What difficulties do research software developers face with code review?

In addition to the benefits of code review, the previous section identifies some of the difficulties in performing code code review. This research question seeks to understand whether those difficulties also hold for research software. We need to collect information about the challenges and barriers research software developers face.

RQ4: What improvements to the code review process do research software developers need?

Similar to the improvements to tooling, processes, and investments commercial and open-source projects have made relative to the code review process (as discussed in the previous section), this question seeks to understand changes that could help code review for research software. To answer this question we need to gather information about the gaps in the process and the desired improvements.

3.4 Methodology

For this study, we gathered data from two sources: interviews and surveys. This section describes the survey/interview questions, the methods for collecting data, and the data analysis process.

3.4.1 Survey/Interview Design

Using the research questions defined in Section 3.3, we enumerated a series of interview and survey questions to gather the necessary information. We piloted the initial set of questions with four research software developers to ensure the questions were clearly

understandable. Based on their feedback, we made the following updates to the questions: (1) rephrased some questions to make them more easily understandable by research software developers, (2) removed software engineering terminology that may be unfamiliar to potential respondents, and (3) rearranged the question flow to be more logical.

Figure 3.1 & 3.2 contain the final questions (after updates from the pilot) organized by research question. The primary difference between how we asked these questions in the interviews and in the surveys is in the interviews, we asked all questions as open-ended questions (except for Q5). We then derived the multiple choice answers listed after each question in Figure 3.1 & 3.2 based on the responses to the survey. In addition to asking questions related to the four research questions, we also included general questions to describe the demographics of the respondents. These questions help characterize the sample to provide additional confidence in the results.

3.4.2 Data Collection

First, we conducted interviews of research software developers. Then, to reach a broader audience, we conducted a survey. The following subsections detail each data collection method.

Interviews

As part of a summer internship at the National Center for Supercomputing Applications (NCSA), the first author was able to recruit research software developers at NCSA for interviews. In addition, he was able to interview some participants from the annual Einstein toolkit meeting, held at NCSA. The interviewer used the questions in Figure 3.1 & 3.2 as an interview guide, but adapted based on the responses of the interviewee, i.e. reordering questions as needed.

Survey

After completing the interviews, we then encoded the questions shown in Figure 3.1 & 3.2 into the Qualtrics survey tool. We started by targeting Computational Science and Engineering (CSE) projects. Then, we broadened the scope to include a broader audience of research software developers so we could gather input from a diverse set of respondents.

To reach a broad sample within the target population, we employed a number of solicitation methods. First, we sent the survey to contributors from the projects represented by the interviewees (but excluded the interviewees). Second, we advertised the survey at the 2017 International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering[1]. Third, a collaborator sent the survey to a mailing list of research software developers in the UK. Fourth, a survey respondent shared the survey on the Research Software Engineers Slack channel. Fifth, we advertised the survey in the monthly newsletter of the Better Scientific Software (http://bssw.io) organization. Finally, we asked respondents to forward the survey invitation within their own networks. As a result of this solicitation approach, we cannot estimate the number of people who received the invitation.

3.4.3 Data Analysis

Due to the length of the survey and to ensure we had adequate data for analysis, we retained the 62 surveys that were at least 90% complete. Then, the first author transcribed the audio recordings from the 22 interviews. Because we performed the interviews before the survey, we asked the interviewees to refrain from taking the survey. We did not want to bias the results by having two sets of answers from the same person. Therefore, there is no overlap between the interview and the survey participants. Together the interviews and surveys produced data from 84 unique participants for the analysis.

We used SPSS to analyze the quantitative data and NVivo for the qualitative data.

[1] https://se4science.github.io/SE-CODESE17/

Figure 3.1: Survey Questions Part I

General Questions

Q1 Which CSE projects are you most actively involved in? (Optional)

Q2 How many years have you worked on CSE projects? (Less than 1 year, 1 years to less than 5 years, 5 years to 10 years, More than 10 years)

Q3 Please choose your roles on the CSE project? (add new code, report bugs, fix bugs, maintain project infrastructure, make strategic decisions about direction of projects, others)

Q4 Do you receive financial compensation for your participation in the project? (Yes, No)

Q5 What is the balance between code you review and code you ask others to review? (1 - Only a code reviewee, 2, 3, 4 - Equally a code reviewee and a code reviewer, 5, 6, 7 - Only a code reviewer)

RQ1: How do research software developers perform code review?

Q6 Please describe the code review process on your project.

Q7 What portion of the overall code commits in the project undergo review? (Less than 10%, 11% - 25%, 26% - 50%, 51% - 75%, More than 75%)

Q8 On average, how many people review code in a given month for the project?

Q9 What factors do you consider when deciding whether to accept a code review request? (Leave blank if you have not reviewed code)

Q10 How much code is typically reviewed at one time?

Q11 On the project, how many hours per week, on average, do you spend reviewing other contributors code? (Less than 1 hour, 1 - 5 hours, 6 - 10 hours, 10 - 15 hours, 16 - 20 hours, More than 20 hours)

Q12 How long, on average, does it take for a first response after a review request? (Less than 1 hour, Less than 1 day, 1-3 days, 4 - 7 days, More than 7 days)

Q13 How long, on average, does it take for a final decision after a review request? (Less than 1 hour, Less than 1 day, Less than 1 week, Less than 1 month, More than 1 month)

Q14 What types of problems do code reviews identify?

Q15 Please explain any positive experiences you have with code review.

Q16 Please explain any negative experiences you have with code review.

Figure 3.2: Survey Questions Part II

RQ2: What effect does code review have on research software?

Q17 Do you agree or disagree with the following statement: Code review is important in the project? (Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree, Strongly agree)

Q18 Please explain why or why not?

Q19 Do you agree or disagree with the following statement: Code review helps improve the code? (Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree, Strongly agree)

Q20 Please explain why or why not?

Q21 Do you agree or disagree with the following statement: Code review helps decrease code complexity? (Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree, Strongly agree)

Q22 Please explain why or why not?

RQ3: What difficulties do research software developers face with code review?

Q23 What aspects of the code review process are challenging?

Q24 What barriers do reviewers face when reviewing code?

RQ4: What improvements to the code review process do research software developers need?

Q25 Is there anything missing from the code review process?

Q26 How could the code review process be improved?

We used a standard qualitative analysis approach to code the survey and interview data, as follows. First, each author individually coded the qualitative responses. Then, we compared these coding activities, consolidated items that had similar codes, and identified items where we disagreed. We then discussed and resolved each of the disagreements. Examples of the most common types of disagreements include: (1) items where we code a response with semantically different codes, which we resolved by discussing and agreeing on the most suitable code (or codes), (2) items where we code a response with differently worded but semantically similar codes, which we resolved by choosing one of the terms. In the end, we resolved all disagreements and arrive at a final agreed-upon coding result.

3.5 Results

We organize this section around the four research questions. Before discussing the specific results, we first provide an overview of the demographics to characterize the sample. Throughout this section, the question numbers refer to the survey questions in Figure 3.1 & 3.2. For the free response questions, our analysis could assign multiple codes to an individual answer. Therefore, in many of the analyses below, the sum of the responses is larger than the number of participants.

3.5.1 Demographics

The answers to Q1 showed participants represent at least 45 research software projects, 13 participants chose not to reveal their project name for privacy reasons. The answers to Q4 indicated that 72% of respondents received some sort of financial compensation for participating in their projects. The answer to these questions indicates that the participants come from a wide variety of projects and have a strong tie to those projects because of the financial compensation.

The distribution of responses to Q2 (Figure 3.3), indicates most participants had at least 5 years of experience working in research software. Just under 1/3 had more than ten years experience. Only a small number had less than one year. This distribution suggests that the participants had appropriate experience and knowledge to provide valid answers to the questions.

The distribution of responses to Q3 (Figure 3.4), indicates that the study participants assume different roles within their respective projects. It is interesting to note that many participants play multiple roles.

Finally, the answers to Q5 (Figure 3.5) show the respondents overwhelmingly participate both as a code reviewer and reviewee. Only a small number of respondents act exclusively as either a reviewer or as a reviewee. An analysis of the raw data indicated that participants with more experience tend to review more code, while those with less

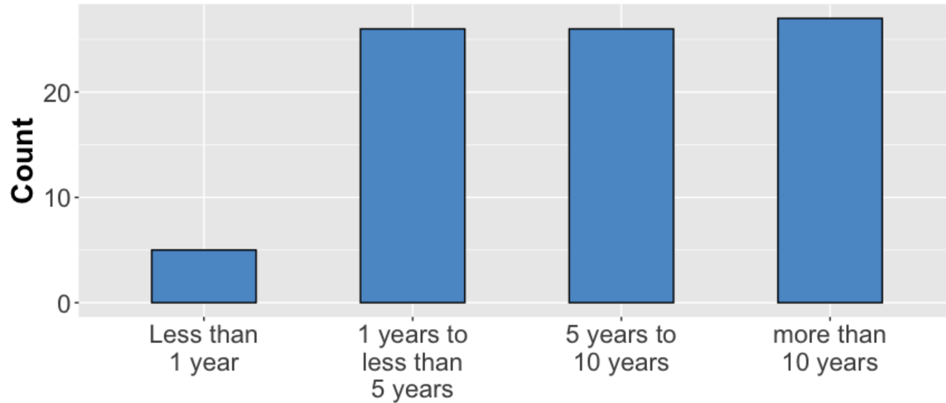

Figure 3.3: Number of years worked on research software

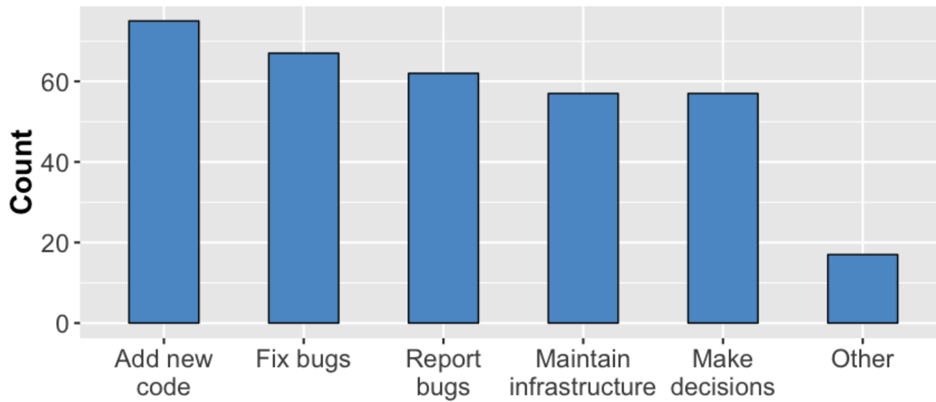

Figure 3.4: Respondents' role on project

experience tend to write more code. Therefore, the study participants have appropriate expertise both with reviewing code and with receiving feedback from reviews to provide valuable insights into the code review process.

3.5.2 RQ1: How do research software developers perform code review?

The following text discusses the overall practices of code review along with the respondents' experiences associated with the code review process.

Overall practices

In response to Q6, respondents described their code review process. Overall, the respondents' projects typically follow an informal code review process. Sixty-one

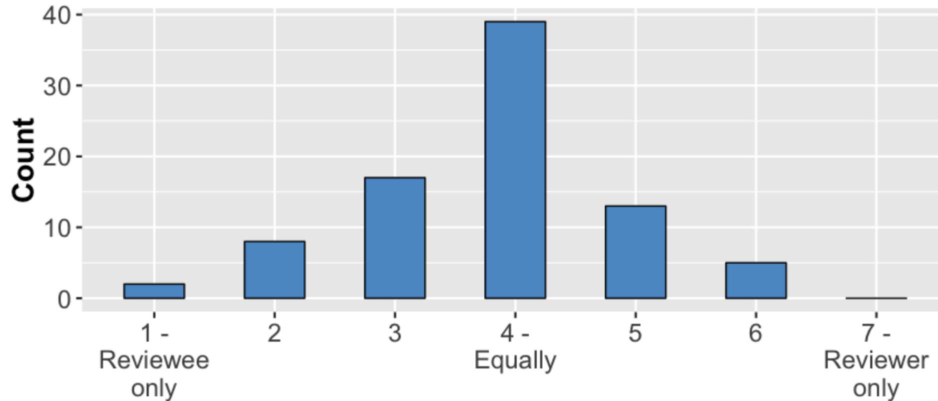

Figure 3.5: Balance as a reviewee and reviewer

respondents indicated they initiate code review with their peers through pull-request on GitHub, Bitbucket, or GitLab. Respondents from larger projects combine an internal ticketing system with the pull requests. The responses varied as to how many people had to review each change or pull-request (anywhere from one to three) before merging into the main branch. In some cases, small changes or bug fixes from experienced or core developers could bypass the review process entirely.

The responses to Q7 (Figure 3.6) show that, in the projects represented by the respondents, more than 75% of the code undergoes peer review. This result is consistent with the previous research on the percentage of code typically reviewed in commercial and open source software [8].

The responses to Q8 (Figure 3.7) shows a wide variety in the number of people that participate in code review. A further analysis of the raw data suggests that in the larger open-source research projects, only core developers perform code review, while in the smaller projects, almost all of the developers perform code review. This observation makes sense as participants in smaller projects have to take on more tasks.

Acceptance of review requests

The results from Q9 (Figure 3.8) show which factors affect the participants decision to accept a code review request. The most common factor is that the code follows **coding**

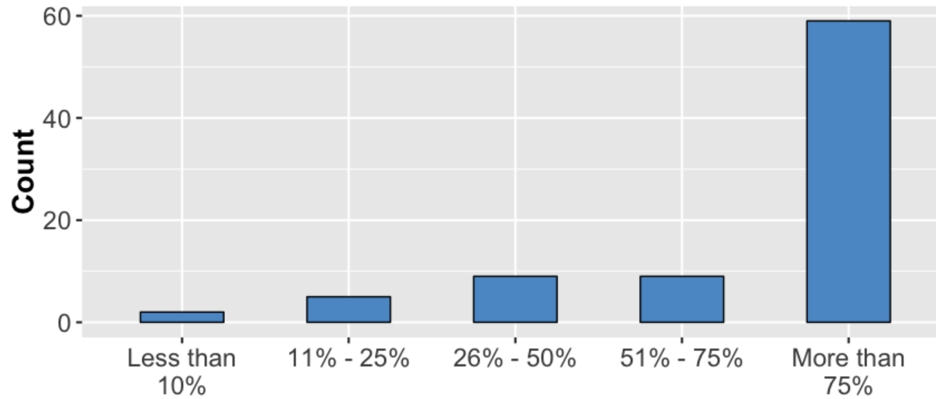

Figure 3.6: Percentage of code undergo review

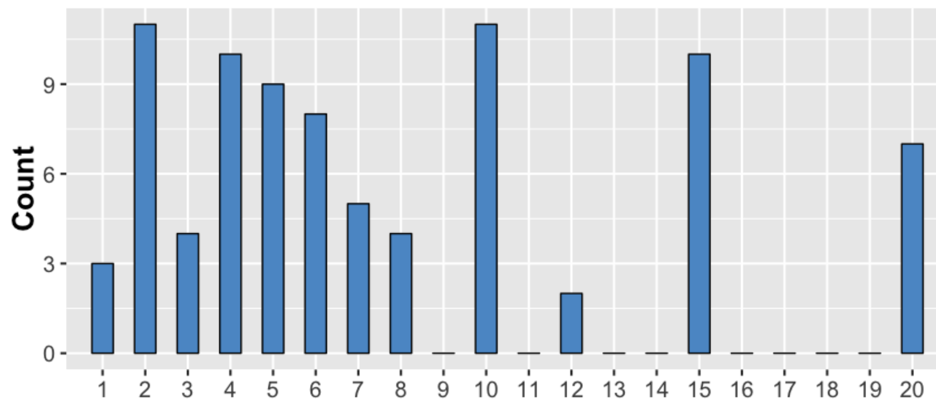

Figure 3.7: Number of reviewers

standards. As one respondent stated, *"all changes need proper style (PEP-8 compliant at minimum) and need to pass the test-suite. Bug fixes just need to fix the known issue and not break any APIs, and potentially add relevant new tests. New features need to be fully documented and have tests."*

The second most common factor in deciding whether to accept a review request is **domain knowledge**. Reviewers want to *"...have knowledge of the project, of the language, or the intended functionality."* They also want to know *"the relevance of the review and whether or not, I am qualified to do the review"*. The belief that *"if I know enough about the edited code to make a good judgment"* summarizes the role of domain knowledge in the decision to accept a review request.

Other factors that influence the decision to accept a review request were **correctness** of the code and whether its **functionality** addresses the project needs. For example, *"does it [the change] add the functionality that we want and does it break any of the existing functionality."* In addition, some participants always accept a code review request. For any requests that are outside their expertise, these participants then refer the review to someone more appropriate.

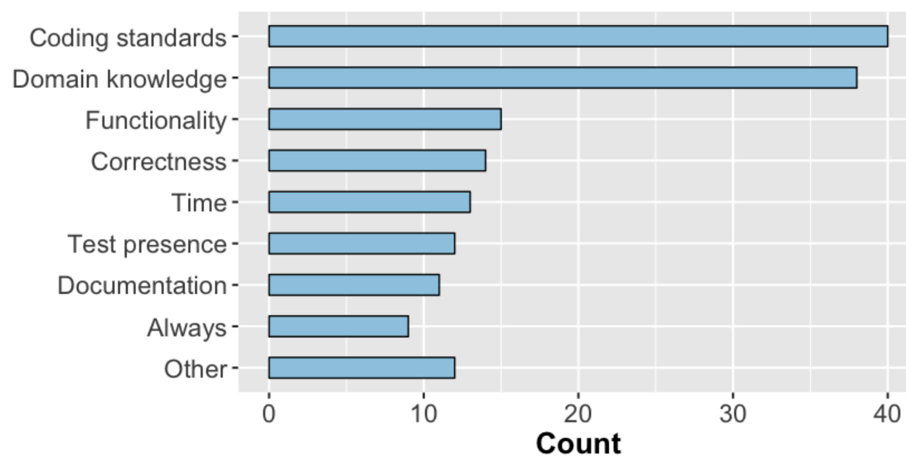

Figure 3.8: Factors to accept code review request

Amount of Code Reviewed

The answers to Q10 varied from project to project. Because the survey question did not specify the unit of measure, the respondents took three perspectives on describing how much code they reviewed at one time.

The first perspective, mentioned by 37 respondents, used files as the unit of measure. One participant said the amount of code reviewed *"varies greatly, from 1 file to 80 files. Most reviews involve about 5-20 files."* These files usually contain a subroutine, module, or a small code segment but *"this depends on the size of the merge request, though we encourage small merge requests to make the job manageable."*

The second perspective, mentioned by 21 respondents, used lines of code (LOC) as the unit of measure. One said *"this [the size of the reviewed change] varies widely. Student PRs are often less than 100 lines, but need many iterations. Developer commits are sometimes*

much longer, but need less work." Typically it is *"50-200 lines of code plus associated tests/documentation (∼500 lines total), but occasionally ranging from 5 to 10,000 lines."*

The third perspective, mentioned by 26 participants, used pull-request (PRs) as the unit of measure. *"[A] PR is usually one physics module, new problem type, etc. It's generally around a few hundred lines of code at a time, max."* Participants review from *"one PR, preferably smaller than 1000 LOC or at least broken into smaller commits, but sometimes large changes"* to a few PR per day.

Time Spent Reviewing Code

The responses to Q11 (Figure 3.9) show that most respondents spend one to five hours per week on code review. An additional 1/3 of respondents spend less than one hour per week. Still fewer respondents spend more than five hours per week. This result is similar to previous results about the amount of time spent in code review for commercial and open source software projects [8].

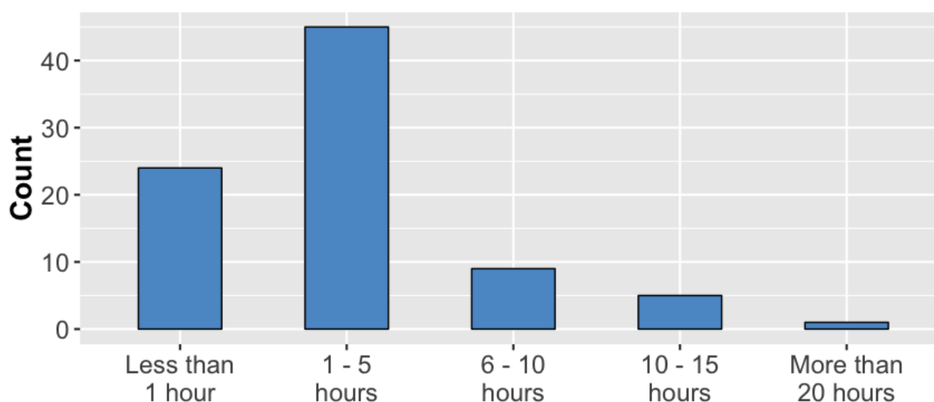

Figure 3.9: Time spent on code review

The responses to Q12 (Figure 3.10) show how long it takes a code author to get the first feedback on his or her code submission. Interestingly, 43% of the respondents said it takes less than a day for a first response. This result is surprising given that conducting code review is not the primary job of research software developers, nor do they receive incentives for performing code review. Approximately 40% of the respondents indicated the

response time was 1-3 days. This result seems reasonable given that research software developers have other tasks related to their own work and may not prioritize reviewing other's code quite as highly.

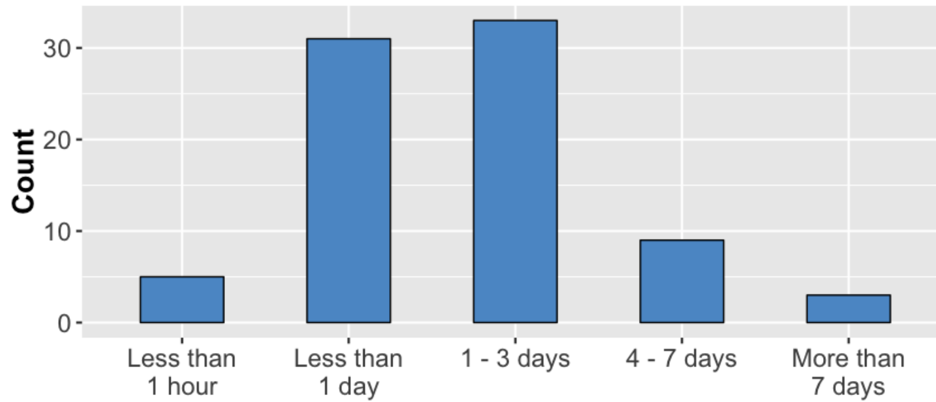

Figure 3.10: Time for a first response

The responses to Q13 (Figure 3.11) provide insight into the overall amount of time taken to reach a final decision on a code submission. Approximately 45% of respondents indicated the project reached a final decision in less than a week. For very small changes or bug fixes, that time was even shorter (less than a day). Overall, 93% of the respondents indicated projects reach a decision in less than a month. The results from the interviews provided some insight into the time required for a final decision.

According to one, "Depends on the size of the contributions–small things could be accepted in a day or two. Large things could take weeks or even months." The interviewees indicated that longer review times were often not the result of a review request sitting idle. In cases where the change is larger or represents a new feature, it may take longer for reviewers to provide proper feedback. According to another, "This [the review time] is incredibly variable, and depends on the reviewers, what changes are requested, and the time it takes for the person issuing the PR to answer. I'd say that the median time is a week, with the mean being higher because some PRs take weeks to months due to slow response times."

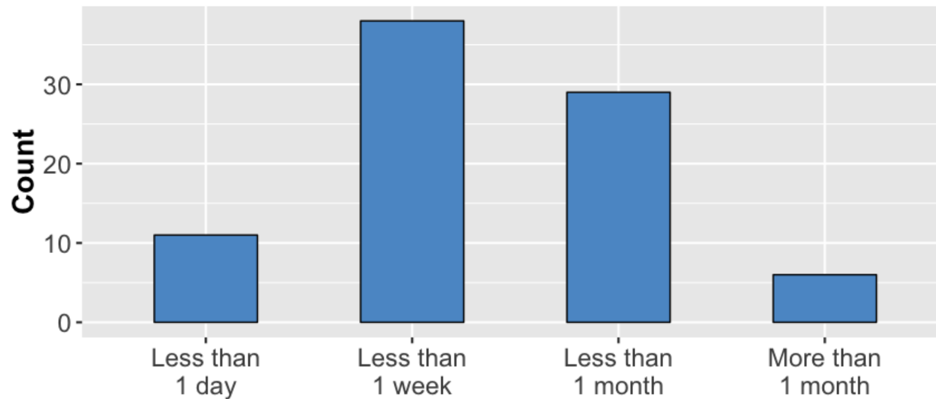

Figure 3.11: Time for a final decision

Problems Identified

The responses to Q14 indicated code reviews help research software developers identify many problems in the code. Figure 3.12 shows three types of problems reviewers identify. Most of the respondents identified problems related to **software quality** and **code mistakes**. Code review identifies *"all sorts of stuff. Poor code, actual bugs, poor usage of macros, code with too narrowly-defined purpose that could easily be generalized."* Similarly, code reviews *"identify corner cases that the author may not have seen, performance issues that might be introduced, breakages with other pieces of the software, algorithmic problems, typos, lack of comments/documentation."*

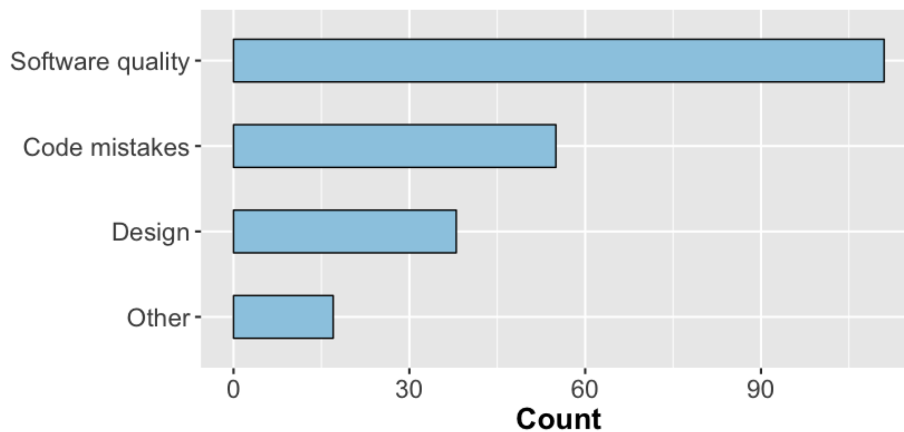

Figure 3.12: Problems code review identifies

Positive Code Review Experiences

The responses to Q15 (Figure 3.13) identified five categories of positive experiences. Relative to the most common positive response, **knowledge sharing**, one participant said *"I find it [peer code review] to be a very cooperative process in which suggestions are welcomed and coders are looking for guidance. In a big project it is rare that anyone understands the whole picture. We rely on each others experience with their part of the project. It can lead to more complete understanding of the task."* Similarly another participant said *"It [peer code review] leads to design discussions happening that would not have happened otherwise. It makes the team more knowledgeable about what work is happening on the project and how people are going about it."*

The second most common positive experience was **improved code quality**. One benefit to code quality was *"people found mistakes in code that I wrote, that I would have missed and only found out about much further on the validation process."* Code review also results in *"much better code and a better understanding of different parts of the code."* This benefit is not limited to new project members because *"everyone's code is better with peer review ... [including] the founders of the project who have 30+ years of programming experience."*

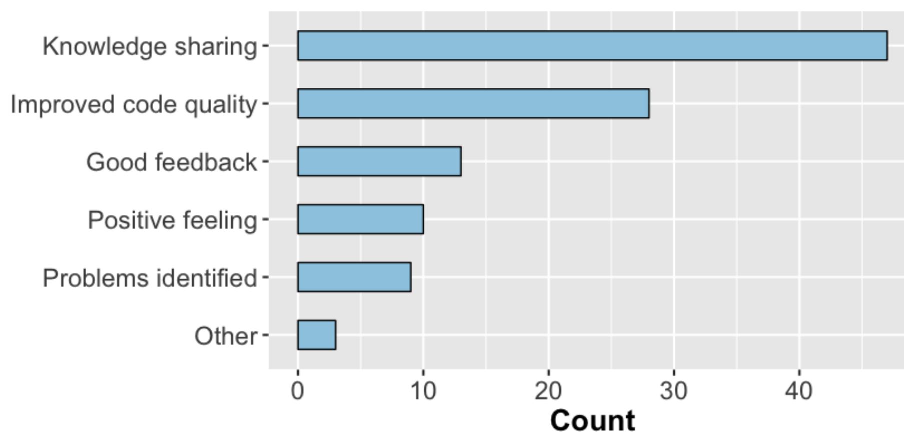

Figure 3.13: Positive experiences with code review

Negative Code Review Experiences

The responses to Q16 (Figure 3.14) resulted in seven categories of negative experiences. The top two negative experiences are the code review process **takes too long** and code authors **misunderstand criticism**.

Regarding the fact that the code review process **takes too long**, one respondent said *"it [peer code review] can be long and time consuming for very small changes, as the process must be followed for even a single character change if it affects results."* There are also problems when the *"review process gets stalled while nit-picking irrelevant details"* or the *"reviewer holds contribution hostage while asking to address unrelated problem at the same time."*

Respondents are also concerned that the code authors will **misunderstand criticism**. This concern makes reviewers less willing to provide criticism and concerned with how the criticism might affect team dynamics. As one respondent noted *"I'm afraid the reviewee could be offended when I write 'this is wrong!' I try to write something more along the lines of 'this should be x instead.' It's frustrating sometimes to overthink this. I would bring up my doubts about not fulfilling all three items in the pull request checklist, and they wouldn't be addressed."*

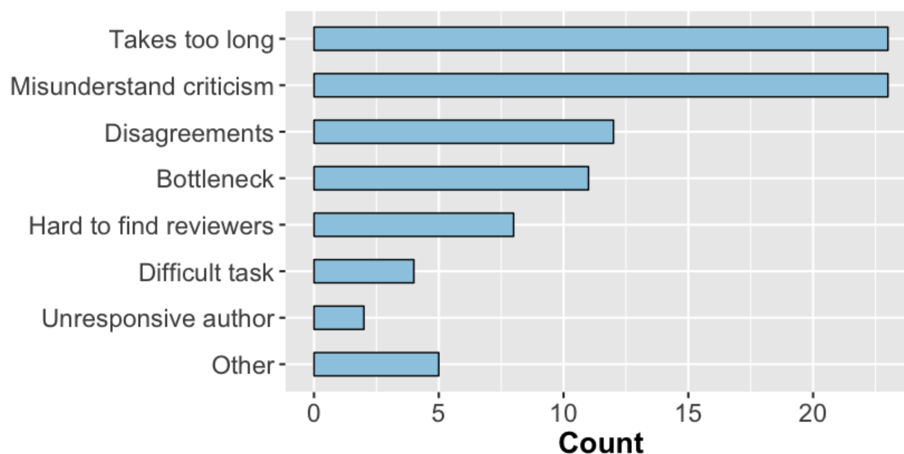

Figure 3.14: Negative experiences with code review

3.5.3 RQ2: What effect does code review have on research software?

The results of Q17 showed that 74 of the 84 participants **strongly agreed** that code review is important for their project. Perhaps this result is not surprising given that these people chose to participate in a survey about code review practices. In response to Q18, the participants gave four primary reasons for the positive responses (Figure 3.15). In terms of **improving code quality**, participants said *"reviews are a way to improve the code and learn, without them bugs would proliferate and code quality would decrease"* and *"it's a means of improving the quality of the software, and of improving the individuals who write it"* In terms of **knowledge sharing**, *"code review helps ensure that at least two people have always looked at each piece of code, spreading out expertise. Additionally, it is a forum for learning from each other"*.

One of the few participants who did not think code review was important suggested that *"requiring code review as a compliance activity is a waste of time. Since that is our project's policy, I view code review as a box to be checked"*.

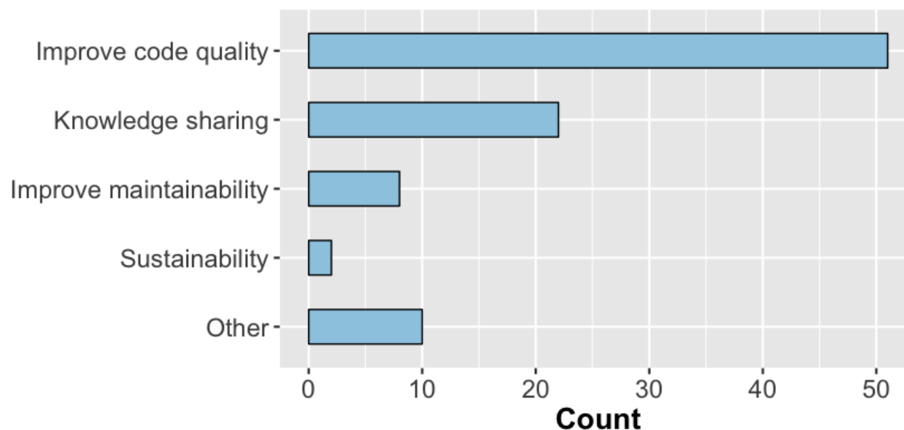

Figure 3.15: Explanation of why code review is important

The results of Q19 showed that 77 out of the 84 participants **strongly agreed** that code review improves code. In response to Q20, the participants gave nine reasons why they believed code review improved their code (Figure 3.16). The most popular reason participants gave was that code review helped with **correctness**. As one respondent said

"If you've written code yourself, it's hard to see the assumptions you've made. Others can spot these and ask you to clarify, also spot your mistakes." Code review also improves code because it helps **improve readability** by *"...make[ing] the codebase more uniform and improves the quality of the code"*. In addition, having **more eyes** look at the code is beneficial because *"having a second pair of eyes often catches issues the author missed or didn't consider"*.

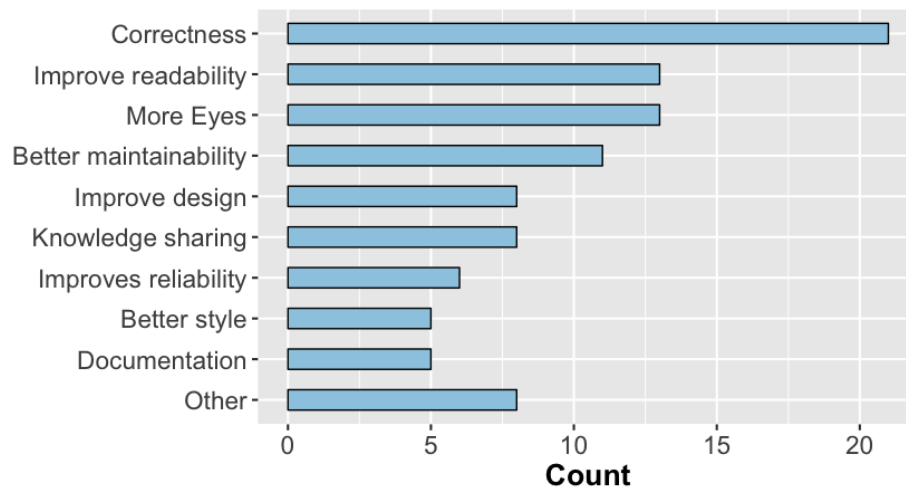

Figure 3.16: Explanation of why code review helps improve the code

The responses to Q21 (Figure 3.17) shows that most participants agreed that code review helps decrease code complexity. In response to Q22, participants explained their answer regarding the impact of code review on code complexity. For the participants that thought code review helped reduce complexity, the explanations mirrored those in Figure 3.16. For example, code review *"decreases complexity by solving problems using cleaner strategies, but may increase complexity in the near term by forcing it to handle corner cases that would not otherwise be discovered until later."*

Conversely, some participants did not see code review as a means to reduce complexity. One participant who did not see any relationship between the two said *"code review often has no effect on code complexity whatsoever. It rare that someone skimming code for a few seconds will think of something drastic that the developer who spend days on*

it did not". Another participant addressed the inherent complexity of some software saying *"code sometimes needs to be performant; a less elegant solution that increases performance by > 5% is better for us than an elegant solution"*.

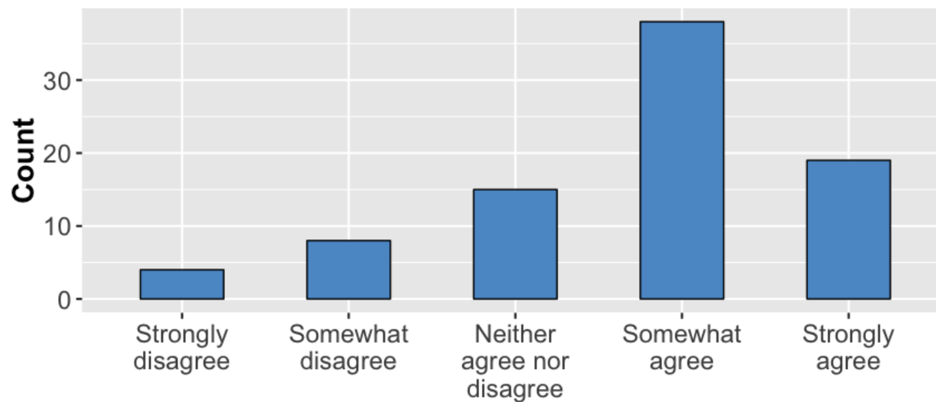

Figure 3.17: Code review helps decrease code complexity

3.5.4 RQ3: What difficulties do research software developers face with code review?

The following text discusses the respondents perspectives on challenges and barriers in the code review process.

Challenges

The responses to Q23 (Figure 3.18) produced four types of challenges. Overwhelmingly, the most common challenge is **understanding code**. For some, *"understanding code is significantly harder than writing code."* Understanding someone else's code requires time because the reviewer has *"to work out exactly what added code is doing so that you can evaluate it."* In addition, reviewers have difficulty reading and understanding large changes.

The second most common challenge is **understanding the system**. The reviewer needs a broader context than the specific change being reviewed. The difficulty arises from *"large diffs, diff is hard to read. Need good knowledge of existing code."* In addition, often *"the code base is broad so the reviewer may know little about what the code is attempting to*

accomplish."

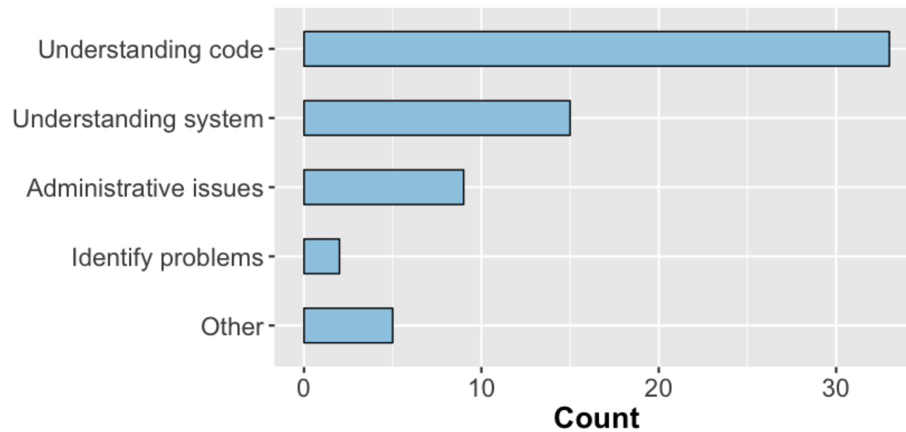

Figure 3.18: Challenging aspects of code review

Barriers

The responses to Q24 (Figure 3.19) produced six types of barriers. Overwhelmingly, the most common barrier is **time**. Code review takes *"time away from the work you're 'supposed' to be doing."* The context switch also costs time because *"setup of the code under review also means making a commit of your local work and perhaps blowing away your own test data."*

The second most common barrier relates to the **phrasing of comments**. For example, *"often our reviewers don't dare to criticise code, and need lots of encouragement to do so. It sometimes helps to have pairs of reviewers instead of assigning them individually."* Reviewers are also concerned about how their comments are perceived because *"there is no tone in written comments (even though I could use emojis!) and I'm afraid the reviewee could be offended when I write 'this is wrong!' I try to write something more along the lines of 'this should be x instead."'*

Another barrier is **finding the right people**, that is *"people with both domain knowledge and the coding knowledge."* It is difficult to find people who want to devote a lot of their time to reviewing code from other developers because (1) There are not many people who possess both domain knowledge and software development knowledge and (2)

Researchers like to get recognition for their work [22, 23], but do not receive any credit for performing code reviews.

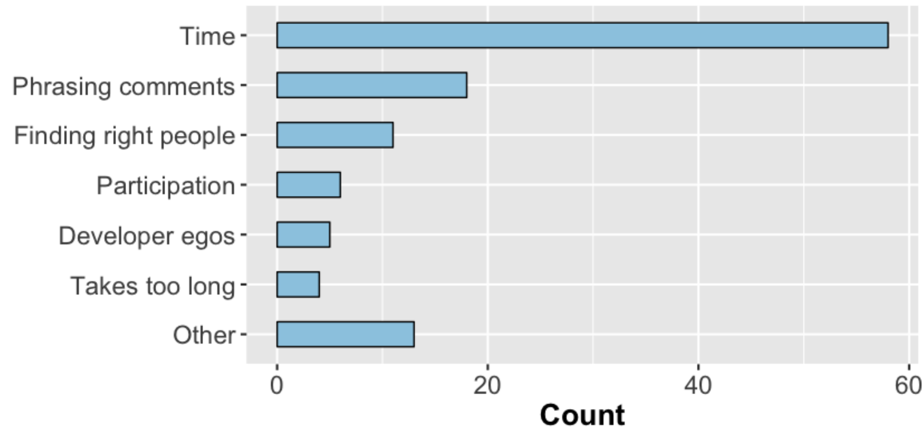

Figure 3.19: Barriers reviewers face when reviewing code

3.5.5 RQ4: What improvements to the code review process do research software developers need?

Because Q25 and Q26 produced similar responses, here we only discuss Q26. The responses to Q26 (Figure 3.20) produced in seven ways to improve the code review process. The most common way, **formalizing process**, is likely due to the informal process followed by most research software developers. One participant suggested using *"a more formal structure of at least one science review followed by one technical review. It's currently a bit of a free-for-all."*

Next *automatic tooling* could relieve some of the burden on the reviewers by making *"it easier to switch between forks and branches in operation."* Besides, *"automated tools can save times for both review and reviewers"* which *"include automatic analysis to release pressure from the reviewers."*

Finally, there is a need for **more people**, **better incentives**, and **more training**. One participant summed it up as *"the main problem is the number of people actually doing it, and the amount of their time. Having said this: besides payment (often not possible) some other kind of reward might help with that."* Another indicated the need for *"training*

more code reviewers so as to increase the reviewer base."

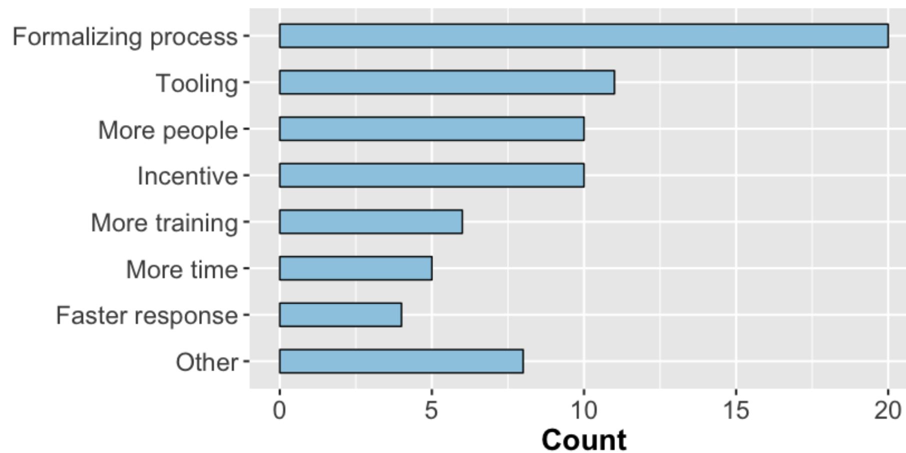

Figure 3.20: Improvement areas in the code review process

3.6 Discussion

In this section, we discuss key insights relative to each of the research questions.

RQ1: How do research software developers perform code review?

Research software developers generally employ an informal code review process. This process identifies many types of problems ranging from typos to functionality. Participants have both positive and negative experiences with code review. Though they do not like when code authors misunderstand their criticism nor the time-consuming nature of the code review process, they do find the knowledge sharing and improved code quality to be positive aspects of the code review process.

RQ2: What effect does code review have on research software?

Code review has an overall positive effect on research software. Participants found code review to be very important for their project. However, the participants did not think code review had much effect on reducing code complexity because sometimes complex code is necessary to increase performance. Overall, it is clear that code review greatly improves code quality by increasing correctness and clarity, which makes the code more readable and

maintainable over time.

RQ3: What difficulties do research software developers face with code review?

There are many difficulties associated with the code review process in research software development. The most common difficulty reported by respondents is finding time to perform code reviews. Another major challenge is understanding other people's code. In addition, sometimes understanding the whole system adds difficulty to the code review process.

RQ4: What improvements to the code review process do research software developers need?

There are several potential improvements to the code review process for research software. The most important improvement is for projects to formalize the review process by including more people, more training, and providing compensation for performing code review.

3.7 Threats

This section describes the primary validity threats to this study.

3.7.1 Internal Threats

The primary threat to internal validity is whether participants understood the software engineering concepts in the same way we intended them. Because the members of the target survey population are not traditional software engineers, it is possible that they lacked the necessary knowledge to properly answer the questions. Because the participants used many appropriate software engineering terms to describe their practices, we believe they do have adequate software engineering knowledge to participate in this study. Therefore, this threat is minimal.

3.7.2 External Threats

It is possible that the study participants are not representative of the population of research software developers. To reduce this threat, we recruited participants from different countries and different projects. While it is clear that the participants are all research software developers, some of the responses suggest that they may be more interested in code review than the average research software developer. In addition, they took time to answer a survey about code review. Therefore, the responses may be biased towards those developers who are already predisposed towards the use of code review. Such a skew in the population would produce results irrelevant to many scientific software developers.

3.7.3 Construct Threats

The primary construct validity threat is the participants may misunderstand the questions. We took great care in writing the survey questions and verified them by expert research software developers and software engineering researchers. In addition, we explained the questions to the interview participants without biasing them so they could used their own judgment to respond.

3.8 Conclusion

In this paper, we report insights about peer code review in research software based upon the responses from 84 survey and interview participants. We also report the effects of the code review process on their projects, difficulties they face in performing code review, and recommendations to improve the process. Considering that our survey targeted a subset of software projects, research software, and previous experience shows that these types of projects are often less advanced in the software engineering process, we were still able to attract a good number of survey respondents.

In conclusion, code review is an essential and important practice for research software development to produce high-quality, trustworthy software. But peer code review is not the

only process required for producing high-quality software. Other software engineering practices, such as software testing, use of proper software metrics, and a suitable development process also help developers produce reliable results. Previous studies show that research software developers have positive perceptions about adopting different software engineering practices such as software development process and software metrics [22, 23]. That said, based on the results of this study, we can conclude that peer code review is a useful software engineering practice to help developers of research software produce high-quality results and maintainable software. Though peer code review presents some challenges and barriers to being established as a core practice in research software setting, it can improve research software in many ways and overcome challenges that cannot be addressed by testing or other software quality practices.

Building on the findings of this paper, our next step is to work more closely with research software projects to better understand how these results apply in practice. By interacting directly with research software developers as they write software, we can conduct direct observational studies to gather information in real-time rather than relying on the type of retrospective reports that we obtained from our interviews and surveys. The results from these direct case studies will help to expand the findings of the interviews and surveys and provide additional useful and practical insights for research software developers.

CHAPTER 4

A CASE STUDY OF TESTING RESEARCH SOFTWARE

4.1 Introduction

Research software can enable mission-critical tasks, provide predictive capability to support decision making, and generate results for research publications. Faults in research software can produce erroneous results, which have significant impacts including the retraction of publications [51]. There are at least two factors leading to faults in research software: (1), the complexity of the software (often including non-determinism) presents difficulties for implementing a standard testing process and (2) the background of people who develop research software differ from traditional software developers.

Research software often has complex, non-deterministic computational behavior, with many execution paths and requires many inputs. This complexity makes it difficult for developers to manually identify critical input domain boundaries and partition the input space to identify a small but sufficient set of test cases. In addition, some research software can produce complex outputs whose assessment might rely on the experience of domain experts rather than on an objective test oracle. Finally, the use of floating-point calculations can make it difficult to choose suitable tolerances for the correctness of outputs.

In addition, research software developers generally have a limited understanding of standard software engineering testing concepts [40]. Because research software projects often have difficulty obtaining adequate budget for testing activities [70], they prioritize producing results over ensuring the quality of the software that produces those results.

This problem is exacerbated by the inherent exploratory nature of the software [34] and the constant focus on adding new features. Finally, researchers usually do not have training in software engineering [19], so the lack of recognition of the importance of the corresponding skills causes them to treat testing as a secondary activity [69].

To address some of the challenges with testing research software, we conducted a case study on the development of a testing infrastructure for the *ParSplice*[1] research software project. The goal of this paper is to *demonstrate the use of a statistical method for testing research software*. The key contributions of this chapter are:

- An overview of available testing techniques for non-deterministic stochastic research software.
- Implementation of a testing infrastructure of a non-deterministic parallel research software.
- Demonstration of the use of a statistical testing method to test research software that can be a role model for other research software projects.

4.2 Background

In a non-deterministic system, there is often no direct way for the tester (or test oracle) to exactly predetermine the expected behavior. In *ParSplice* (described in Section 4.2.1), the non-determinism stems from (1) the use of stochastic differential equations to model the physics and (2) the order in which communication between the procedures occurs (note however that even though the results from each execution depends upon message ordering, each valid order produces a statistically accurate result, which is the key requirement for the validity of *ParSplice* simulations).

In cases where development of test oracles is difficult due to the non-determinism, some potentially viable testing approaches include metamorphic testing, run-time assertions, and

[1] https://gitlab.com/exaalt/parsplice (LA-UR-20-20082)

machine learning techniques [38]. After describing the *ParSplice* project, the remainder of this section explains these techniques along with their possible applicability to *ParSplice*.

4.2.1 ParSplice

ParSplice (Parallel Trajectory Splicing) [58] aims at overcoming the challenge of simulating the evolution of materials over long time scales through the timewise parallelization of long atomistic trajectories using multiple independent producers. The key idea is that statistically accurate long-time trajectories can be assembled by splicing end-to-end short, independently-generated, trajectory segments. The trajectory can then grow by splicing a segment that begins in the state where the trajectory currently ends, where a state corresponds to a finite region of the configuration space of the problem. This procedure yields provably statistically accurate results, so long as the segments obey certain (relatively simple) conditions. Details can be found in the original publication [58].

The *ParSplice* code is a management layer that orchestrates a large number of calculations and does not perform the actual molecular dynamics itself. Instead, *ParSplice* uses external molecular dynamics engines. The simulations used in *ParSplice* rely on stochastic equations of motion to mimic the interaction of the system of interest with the wider environment, which introduces a first source of non-determinism.

A basic ParSplice implementation contains two types of processes: a splicer and producers. The splicer manages a database of segments, generates a trajectory by consuming segments from the database, and schedules execution of additional segments, each grouped by their respective initial state. Producers fulfill requests from the splicer and generate trajectory segments beginning in a given state; the results are then returned to the splicer. The number of segments to be scheduled for execution in any known state is determined through a predictor statistical model, built on-the-fly. Importantly, the quality of the predictor model only affects the efficiency of *ParSplice* and not the accuracy of the trajectory. This property is important because the predictor model will almost always be incomplete, as it is inferred from a finite number of simulations. The unavailability of the

ground truth model (which is an extremely complex function of the underlying physical model) makes assessment of the results difficult. In addition, this type of stochastic simulation is not reproducible, adding to the difficulty of testing the code. Therefore, in this case study we create a basis for the *ParSplice* testing infrastructure using various methodical approaches and apply the test framework to the continuous integration process.

4.2.2 Metamorphic Testing

Metamorphic testing operates by checking whether the program under test behaves according to a set of metamorphic relations. For example, a metamorphic relation R would express a relationship among multiple inputs x1, x2,.., xN (for N $>$ 1) to function f and their corresponding output values *f(x1)*, *f(x2)*,.., *f(xN)* [15]. These relations specify how a change to an input affects the output. These metamorphic relations serve as a test oracle to determine whether a test case passes or fails. In the case of *ParSplice*, it is difficult to identify metamorphic relations because the outputs are non-deterministic. The relationship between the x's and the f's is therefore not direct but statistical in nature.

4.2.3 Run-time Assertion Checking

An assertion is a boolean expression or constraint used to verify a necessary property of the program under test. Usually, testers embed assertions into the source code that evaluate when a test case is executed. Later testers use these assertions to verify whether the output is within an expected range or if there are some known relationships between program variables. A program fails if an assertion fails during the execution of a test case. In this way, a set of assertions can act as an oracle. In the context of *ParSplice*, assertions can be used to test specific functions, but not to test the overall validity of the simulations, would protect only against catastrophic failures, such as instabilities in the integration scheme. This approach is unlikely to thoroughly validate the code in normal operating conditions, but could help validating individual components.

4.2.4 Machine Learning Techniques

Machine learning is a useful approach for developing oracles for non-deterministic programs. Researchers have shown possibilities of both black-box features (developed using only inputs and outputs of the program) and white-box features (developed using the internal structure of the program) to train the classifier used as the oracle [14] [26]. It is possible to test *ParSplice* with machine learning techniques. For example, we could fake the molecular dynamics (MD) engine with our own model to produce output data to use as a training set and consider the actual output data as a testing set. Due to the amount of effort required to use this approach in *Parsplice*, we determined that it was not feasible.

4.3 Case Study

To implement the testing framework, the first author spent the a summer at Los Alamos National Laboratory working on the *ParSplice* project. The testing framework is based on the Multinomial testing approach (described in Section 4.3.2), implemented using a *progress tracking card* (PTC) in the Productivity and Sustainability Improvement Plan (PSIP)[2] methodology. The testing approach is integrated with the CMake/CTest tool for use in the runtime environment and continuous integration. In this section, we describe the PSIP methodology, the Multinomial test approach, and results that verify the implementation of the testing framework.

4.3.1 PSIP

The PSIP methodology provides a constructive approach to increase software quality. It helps decrease the cost, time, and effort required to develop and maintain software over its intended lifetime. The PSIP workflow is a lightweight, multi-step, iterative process that fits within a project's standard planning and development process. The steps of PSIP are: a) Document Project Practices, b) Set Goals, c) Construct Progress Tracking Card , d) Record Current PTC Values, e) Create Plan for Increasing PTC values, f) Execute Plan, g)

[2] https://betterscientificsoftware.github.io/PSIP-Tools/PSIP-Overview.html

Assess Progress, h) Repeat.

We created and followed a PTC containing a list of practices we were working to improve, with qualitative descriptions and values that helped set and track our progress. Our progress tracking card consists of 6 scores with a target finish date to develop the testing framework. The scores are:

- Score 0 - No tests or approach exists
- Score 1 - Requirement gathering and background research
- Score 2 - Develop statistical test framework
- Score 3 - Design code backend to integrate test
- Score 4 - Test framework implemented into ParSplice infrastructure
- Score 5 - Integrate into CI infrastructure

We were able to progress through these levels and obtain a score of 5 by the end of the case study.

4.3.2 Multinomial Test

The Multinomial test is a statistical test of the null hypothesis that the parameters of a multinomial distribution are given by specified values. In a multinomial population, the data is categorical and belongs to a collection of discrete non-overlapping classes. For instance, multinomial distributions model the probability of counts of each side for rolling a k-sided die n times. The Multinomial test uses Pearson's χ^2 test to test the null hypothesis that the observed counts are consistent with the given probabilities. The null hypothesis is rejected if the p-value of the following χ^2 test statistics is less than a given significance level. This approach enables us to test whether the observed frequency of segments starting in i and ending in j is indeed consistent with the probabilities p_{ij} given as input to the Monte Carlo backend. Our Multinomial test script uses the output file of *ParSplice* as its

input and execute the test and post-processes the results by performing Pearson's χ^2 to assess whether to reject the null-hypothesis.

4.3.3 Results

A key insight from the theory that underpins *ParSplice* is that a random process that describes the splicing procedure should rigorously converge to a discrete time Markov chain in a discrete state space. In other words, the probability that a segment added to a trajectory currently ending in state i leaves the trajectory in state j should be a constant p_{ij} that is independent of the past history of the trajectory. One way to test *ParSplice* would be to verify that the splicing procedure is indeed Markovian (memory-less). However, taken alone, such a test would not guarantee that the splicing proceeds according to the proper Markov chain. A more powerful test would assess whether the spliced trajectory is consistent with the ground-truth Markov chain. A key obstacle to such a test is that this ground-truth model is, in practice, unknown and can only be statistically parameterized from simulation data.

To address this issue, we replaced the molecular dynamics (MD) simulation backend with a simpler Monte Carlo implementation that samples from a pre-specified, Markov chain. That is, we replaced the extremely complex model inherent to the MD backend with a known, given model of predefined probabilities. The task then becomes assessing whether the trajectory generated by *ParSplice*, as run in parallel on large numbers of cores, reproduces the statistics of the ground truth model. In this context, this technique is the ultimate test of correctness, as *ParSplice* is specifically designed to parallelize the generation of very long trajectories that are consistent with the underlying model. Statistical agreement between the trajectory and the model demonstrates that the scheduling procedure is functional (otherwise, the splicing the of trajectory would halt), the task ordering procedure is correct, the tasks executed properly, the results reduced correctly, and the splicing algorithm was correct. What this test does not cover, however, is the correctness of the underlying MD engine. Such a test can be considered the

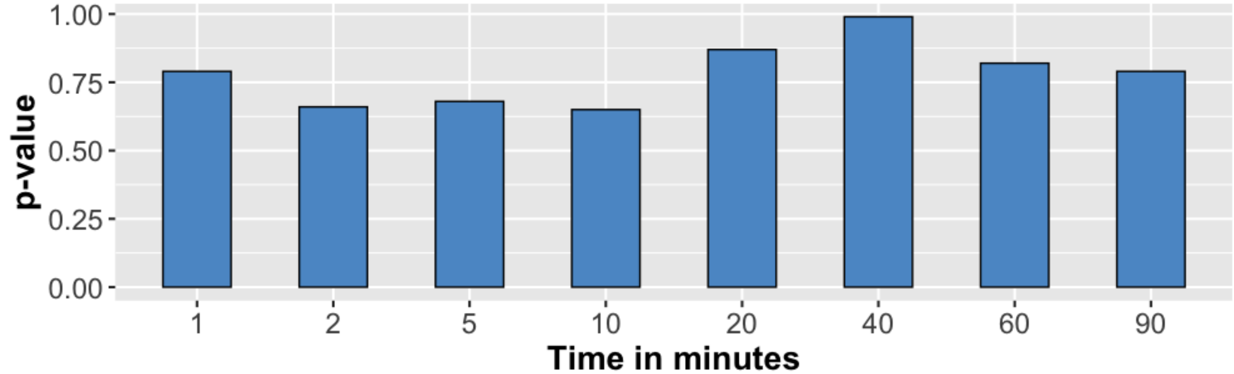

Figure 4.1: p-values obtained by executing *ParSplice* for different times.

responsibility of the developer of that code.

The statistical assessment to test *ParSplice* can be conducted using the Multinomial test approach. Our null hypothesis was that the observed counts generated by *ParSplice* are consistent with the probabilities in the model. If the p-value from the multinomial test is less than 0.05, we reject the null hypothesis and conclude that the observed counts differ from the expected ones. Conversely, if the p-value is greater than 0.05, we do not reject the null hypothesis and can conclude that the test passes. For the sake of verifying our Multinomial test, we ran *ParSplice* in different time frames and observed the result. Figure 4.1 shows the p-values obtained from running *ParSplice* for 1, 2, 5, 10, 20, 40, 60, and 90 minutes. We can see that in all cases, the p-values are greater than 0.05, which indicates that the tests passed during these instances of the execution.

4.4 Conclusion

Testing research software can be difficult due to the inherent complexity of the software (especially when it is non-deterministic) and the lack of formal software engineering training of the developers. In this paper, we describe a case study of the *ParSplice* project in which we followed the PSIP methodology and developed a testing framework to address the difficulties of testing non-deterministic parallel research software. We first considered applying traditional industrial testing approaches. However, the

non-determinism of *ParSplice* made these approaches unusable. Then we identified testing techniques specially designed for non-deterministic software. Once again, those techniques did not fit *ParSplice* (as described in Section 4.2). Finally, we identified a statistical testing approach, Multinomial Testing, that would work for *ParSplice*.

The Multinomial Testing approach is ideal for *ParSplice* given its constraints, i.e. time, non-determinism, and the existing continuous integration system. The lessons learned from this case study can be valuable to the larger research software community because, like *ParSplice*, many research software projects have stochastic behavior which produces non-deterministic results. The approach we followed to develop the test framework can be a model for other research software projects. We plan to extend the testing infrastructure in a more methodological way with as many possible testing techniques installed in the system.

CHAPTER 5

A SURVEY OF TESTING RESEARCH SOFTWARE

5.1 Introduction

Research software is software developed by researchers from a wide variety of domains including, but not limited to, science, engineering, business, and humanities [22]. There are various types of research software that serve different purposes. First, researchers develop software to make predictions about or to better understand real-world processes [40]. This type of research software solves computationally complex or data-intensive problems. For example, researchers develop software for large physical phenomena, such as weapons or medical simulations, that run on high-performance computing machine. Researchers also develop smaller simulations that run on desktop machines or small cluster [46]. Second, research software can provide infrastructure support (e.g., messaging middleware or scheduling software) or libraries for mathematical and scientific programming (e.g., linear algebra or symbolic computing).

The quality of research software is critical. Researchers use research software in mission-critical situations and decision making [40]. In addition, researchers use the results from research software as evidence in research publications [17, 67]. Therefore, it is important that research software have a correct design and implementation. Low quality software produces less trustworthy results and is prone to failure in mission-critical situations. Software quality problems have even caused scientists to retract publications [51]. Therefore, researchers need to develop high-quality software that

produces trustworthy results and functions properly in critical situations.

The term *research software developer* refers to a researcher who develops research software. Research software developers range from researchers who possess little or no software engineering knowledge to experienced professional software developers with considerable software engineering knowledge [40].

Testing is a useful practice for producing high-quality software. Unfortunately, because of its complex computational behavior, it is very difficult to test research software . Research software developers often build software based upon a set of mathematical equations and use mathematical analysis to verify the corrections of the computational model [44, 60]. For example, researchers use research software to determine the impact of modifications to nuclear weapon simulations since real-world testing is too dangerous and not allowed [60].

While testing is a useful practice, there are some technical challenges for testing research software. The first challenge is the lack of test oracles [37]. An oracle is pragmatically unattainable in most of the cases for research software because researchers develop software to find previously unknown answers. Due to the lack of test oracles, research software developers often use judgment and experience to check the correctness of the software. The second challenge is the large number of tests required to test research software using standard testing techniques. Also, the large number of input parameters makes it challenging to manually selecting a sufficient test suite [74]. Finally, the presence of legacy code makes testing research software very challenging [16].

Because of these challenges, research software developers are unlikely to use systematic testing to check the correctness of their code [37, 46, 54]. Even though these developers conduct validation checks to ensure the software correctly models the physical phenomenon of interest [44, 54], there is still a need for testing that identifies differences between the model and the code [24]. In addition, sometimes the reason for limited use of systematic testing results from the testing challenges posed by the software itself [20].

Because of the various challenges to properly and fully testing research software, there is a need to better understand how research software developers actually perform testing activities in practice. Therefore, the goal of this paper is to *better understand the testing process, identify challenges, and provide recommendations on how to improve the testing process in research software development.* To make this high-level goal more tractable, Section 5.2 reviews related work and poses a set of specific research questions. Then, to answer these questions, the paper describes the results from our survey of research software developers.

The key contribution of this paper are:

- An overview of the level of knowledge research software developers have on testing techniques;
- A description of the current testing practices used in the research software community;
- A list of the difficulties of the testing research software;
- An analysis of the compatibility of commercial/IT testing techniques to research software; and
- An identification of areas of improvement in the testing process for research software.

5.2 Research Questions

This section discusses the related work. We use the related work to motivate a series of research questions that will ultimately drive the survey design.

Research software developers often have little or no formal software engineering knowledge [19, 20, 30]. There is a lack of recognition for the skills and knowledge required for software development [35]. These developers are typically unfamiliar with available testing methods [21, 30]. As a consequence, they do not usually have a set of written

quality goals. Researchers even treat software development as a secondary activity. Therefore, to better understand the level of knowledge research software developers possess about software testing and verify the claims from the literature, we pose the following research question:

RQ1: What level of knowledge do research software developers have about testing techniques?

The literature does not provide a clear picture of how research software developers perform testing. Research software developers commonly omit Unit testing because they have misconceptions about the benefits and difficulties of implementing unit tests [16]. Research software developers under-utilize verification testing because they are unaware of the need for it and the methods for applying it [19]. In many cases, research software projects do not even include automated acceptance testing and regression testing [57]. Furthermore, the research software community is lagging behind in the use of available testing tools, at least partially due to the wide use of FORTRAN [19, 47, 67]. Therefore, because the literature does not provide a clear picture, to better understand how research software developers actually perform testing, we pose the following research question:

RQ2: How do research software developers test their software?

Testing research software is very challenging. A previous systematic literature reported testing challenges due to the characteristics of research software and the cultural differences between researchers and more traditional software engineers [40]. The authors subdivided the testing challenges resulting from the characteristics of research software into four categories: a) test case development, b) producing expected test case output values, c) test execution, and d) test results interpretation. Then subdivided the testing challenges resulting from the cultural differences between research software developers and more traditional software engineers into three categories: a) limited understanding of testing concepts, b) limited understanding of the testing process, and c) not applying known

testing methods. Because the SLR was only able to capture and report on challenges actually published in the literature, to better evaluate whether these challenges, and others, happen in practice, we posed the following research question:

RQ3: Why is testing research software difficult?

Based on the SLR described in the previous research question [40], there is little evidence that research software projects use the available testing methods as we describe below. Only a few projects mentioned the use of unit testing [2,17,24,45]. There is only one study that mentioned the use of integration testing [17]. A few other studies mentioned the use of system testing [16,24], acceptance testing [16], and regression testing [17,24,36]. There are a lot of research software projects in existence, but the number that appear in literature about testing is very low [40]. Therefore, to understand whether existing testing methods could be adapted to address this lack of software testing in research software, we pose the following research question:

RQ4: Is it possible to adapt existing testing techniques to test research software?

To better identify ways to advance the testing of research software, there is a need to better understand specific ways to improve the testing process and overcome the testing challenges that exist. Because of the challenges, developers often do not want to write tests. Therefore, addressing the testing challenges can make the process welcoming to research software developer. For example, developing a test oracle is a key challenge for testing research software. Some research software projects have addressed this challenge through creating pseudo oracles, which is code developed separately to produce the intended output given the same input as the original program [19,24,57]. Though creating a pseudo oracle is not commonly applied to all research software projects but this technique may work for some. Another challenge for testing research software is obtaining adequate budget [35,57,70]. Projects may be able to overcome this challenge through

increasing awareness among project decision-makers of the need for providing a budget for testing activities. To identify other approaches for improving the current testing process in research software development, we pose the last research question:

RQ5: What improvements to the testing process do research software developers need?

5.3 Methodology

To answer the research questions described in Section 5.2, we conducted a survey of members of the research software development community. To reach a broad audience and produce generalized result, we decided to conduct the survey. Because surveys are useful in describing the characteristics of a large population. The survey ensures a more accurate sample to gather targeted results in which to draw conclusions and make important decisions. Section 5.3.1 describes the design of the study based on the research questions. Section 5.3.2 explains the process we followed to analyze the results of the survey.

5.3.1 Survey Design

Using the research questions, we enumerated the survey questions shown in Figure 5.1 & 5.2. Note that we include the bold headings that enumerate the Research Questions for clarity in the paper, but we did not include them in the original survey. We first developed questions to gather demographic information. This information helps us understand the respondents' background, determine eligibility for taking the survey, and ensuring the survey reached a broad audience. Then, to answer each research question, we identified a set of survey questions to gather specific information related to the question.

Because we anticipated that many survey respondents may not be familiar with standard software engineering definitions, we defined key software engineering concepts on the survey. For example, we provided definitions for the specific testing methods and techniques included on the survey to address the possibility that a respondent may be

familiar with a concept but unaware of the proper term. After developing the initial survey, we piloted the survey with a few experienced research software developers. Based on their feedback, we reworded some questions for clarity to research software developers. We then deployed the survey via the Qualtrics platform.

We used the following solicitation methods to reach a broad population of research software developers. First, we sent the survey to mailing lists that reach research software developers. Those lists include the prior attendees of the SE4Science workshops[1] along with a custom-made email list of contributors to research software repositories mined from GitHub. Second, we advertised the survey in two Slack channels, one for international research software engineers (*RSE*) and one focused on research software engineers in the US (*USRSE*). The subscribers to these slack channels are research software engineers that range from graduate students to experienced researchers working in academia or research labs. Third, we advertised the survey in the monthly newsletter of Better Scientific Software (BSSw) and the IDEAS-productivity mailing list. The participants in these mailing lists are mainly people from the USA working in national labs on scientific projects and faculty members/graduate students/postdocs from different universities. Fourth, we asked people reached by the above advertising to also forward the survey invitation within their own networks. Because of the solicitation approach we used, we cannot estimate how many people received our invitation.

We initially distributed the survey out August 27, 2019. We left the survey open for a month before closing it.

5.3.2 Data Analysis

We received a lot of responses where many of them were partially complete. The first author manually went through each responses and identified the responses that were at least 90% complete and had the ability to provide insightful evidence to the result. We received both quantitative and qualitative data from the survey. We used the tool SPSS to

[1] https://se4science.org/workshops/

Figure 5.1: Survey Questions Part I

General Questions

Q1 Which research software project are you most actively involved in? (Optional)

Q2 Please choose your roles on the project (Choose all that apply)? [Developer, Architect, Quality Assurance Engineer, Maintainer, Manager, Executive, Other]

Q3 How many years have you worked on the project? [Less than 1 year, 1 year to less than 5 years, 5 years to less than 10 years, More than 10 years]

Q4 Which best describes the current development stage of your project? [Planning/Requirements Gathering, Initial Development/Prototyping, Active Development/Unreleased Software, Active Development/Released Software, Maintenance/No New Development Planned, Other]

Q5 How many developers are currently working on your project? [1 to 5, 5 to 10, 10 to 15, 15 to 20, More than 20]

RQ1: What level of knowledge do research software developers have about testing techniques?

Q6 How confident are you on your knowledge of software testing? [Very Low, Low, Average, High, Very High]

Q7 What is your level of understanding on the testing concepts USED in your project? [Very Low, Low, Average, High, Very High]

Q8 What is your level of understanding on the testing concepts NEEDED in your project? [Very Low, Low, Average, High, Very High]

Q9 List any software testing techniques with which you are familiar. [Free response]

RQ2: How do research software developers test their software?

Q10 Please select one of the items below that most closely resembles your goal of testing on your project. [Level 0 - There is no difference between testing and debugging, Level 1 - The purpose of testing is to show correctness, Level 2 - The purpose of testing is to show that the software does not work, Level 3 - The purpose of testing is not to prove anything specific, but to reduce the risk of using the software, Level 4 - Testing is a mental discipline that helps all researchers develop higher quality software]

Q11 Which of the following testing methods does your team use? (Choose all that apply) [Unit testing, Integration testing, System testing, Acceptance testing, Module testing]

Q12 How often is software testing useful in your project? [Never, Rarely, Sometimes, Most of the time, Always]

Figure 5.2: Survey Questions Part II

Q13 Please select any specific testing techniques you use in your project (Choose all that apply). [Metamorphic testing, Assertion checking, Performance testing, Monte carlo test, Dual coding, Fuzzing test, Backward compatibility testing, Using machine learning, Using statistical test, Test driven development, Input space partitioning, Graph coverage, Logic coverage, Statement coverage, Condition coverage, Branch coverage, Syntax-based testing, Boundary value analysis, Equivalence partitioning, Decision table based testing, State transition, Error guessing, Backward compatibility testing, other]

RQ3: Why is testing research software difficult?

Q14 How complex is the testing process on your project?

Q15 Please explain any barriers or challenges you face to test your project?

RQ4: Is it possible to adapt existing testing techniques to test research software?

Q16 How often does your team apply Commercial/IT testing methods in your project? [Never, Rarely, Sometimes, Most of the time, Always]

Q17 How often do you PERSONALLY apply Commercial/IT testing methods in your project? [Never, Rarely, Sometimes, Most of the time, Always]

Q18 How much value do you see personally in using Commercial/IT testing methods? [Very Low, Low, Average, High, Very High]

Q19 Please explain any challenges to adapt Commercial/IT testing methods to your project. [Free response]

Q20 Please explain any challenges you think that could not be met by Commercial/IT testing testing methods? [Free response]

RQ5: What improvements to the testing process do research software developers need?

Q21 How could the overall testing process be improved? [Free response]

analyze the quantitative data. For any data that are qualitative, we individually coded the text using a tool called NVivo. Then we compared our results and identified the discrepancies. We discussed all the discrepancies on an in-person meeting to solve any disagreements. Finally, we categorized the codes into high-level categories. We used the programming language R to visualize the results into charts.

5.4 Results

In this section, we describes the detailed results from the survey. Based on the analysis of completeness, we identified 120 valid survey responses to include in the discussion of results.

5.4.1 Demographics

Q1-Q5 gathered various demographic information about the survey respondents. The following subsections detail each of these demographics.

Projects

Q1 asked the participants to optionally identify their research software project. Respondents mentioned 66 unique research software projects. Because of their project's privacy policies, 56 respondents did not name their project. All but two project names were unique, with one identified by three respondents and the other by two. This result indicates that the respondents came from a wide variety of research software projects, which allows our overall findings to generalize to a broad audience of research software developers.

Project Role

Because people in different project roles likely have different perspectives on software quality and different experience with testing their projects, the respondent's role is an important demographic. The results of Q2 (Figure 5.3) shows the distribution of respondents is skewed towards technical roles (e.g. Developer, Architect, and Maintainer). Note that because research software developers often hold multiple roles on a project, the survey allowed them to choose multiple responses. Therefore, the sum of the bars in Figure 5.3 is larger than the total number of respondents.

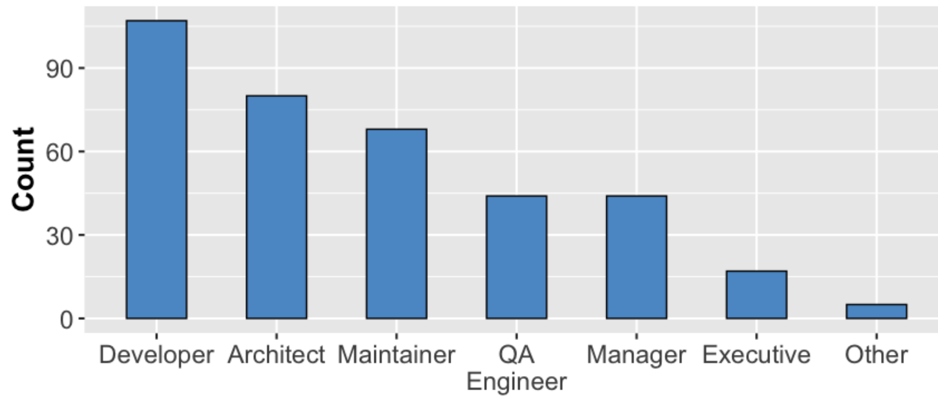

Figure 5.3: Respondents' role on project

Project Experience

A respondent's experience provides insight into whether she or he has enough knowledge to draw upon to provide helpful responses to the survey. The results from Q4 (Figure 5.4) show the large majority of respondents had more than one year of experience in working research software projects. Almost 1/3 of had at least five years of experience, with 1/2 of those having more than ten years of experience. This result shows that, the respondents had enough experience to provide valid responses to the survey.

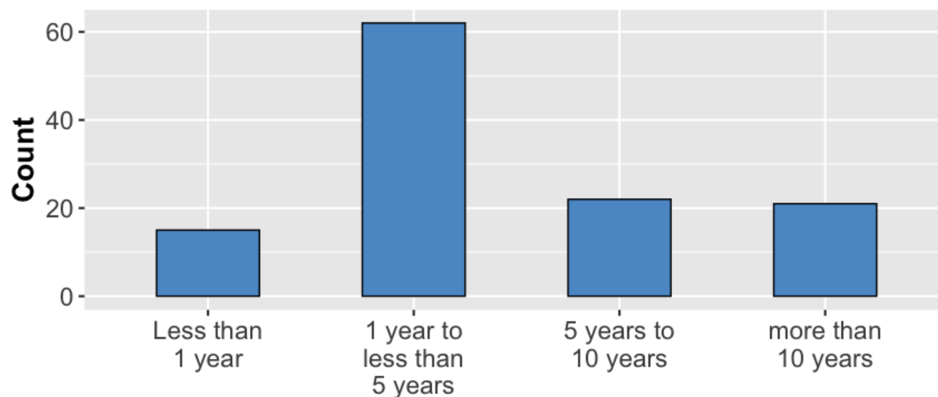

Figure 5.4: Number of years worked on research software

Project Stage

Because different types of testing are relevant at different states of project development, the stage of the respondent's project could impact how he or she answers the survey questions. The responses to Q4 (Figure 5.5) show projects were overwhelmingly at the *released* stage. This result is important because the projects at this stage should have already established testing practices in the project.

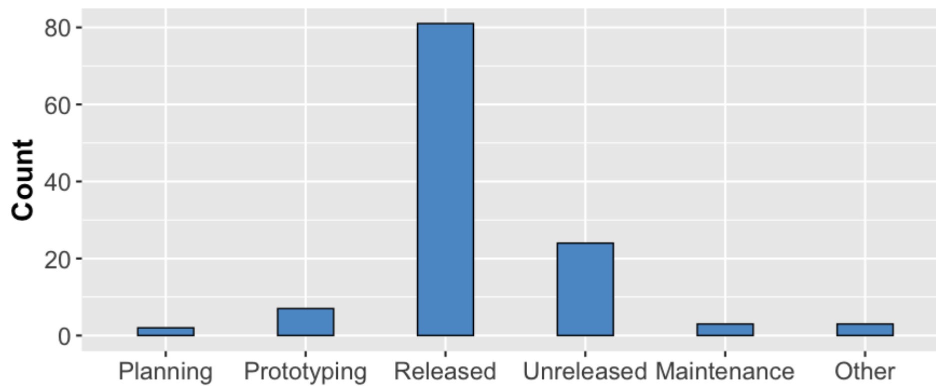

Figure 5.5: Project stage

Project Size

Because teams of different sizes may have different perspectives on the use of software engineering practices [22], we asked the respondents about their project size. The responses to Q5 (Figure 5.6) shows that while the largest group of respondents were on smaller teams, there is also a good distribution of respondents across larger team sizes.

5.4.2 RQ1: What level of knowledge do research software developers have about testing techniques?

Survey question Q6 asked respondents how confident they were in their knowledge of software testing. The results in Figure 5.7 show most respondents indicated they possessed at least an *average* level of confidence about testing knowledge, with more than 1/3 indicating *high* or *very high* confidence in their knowledge. On a positive note, only a few

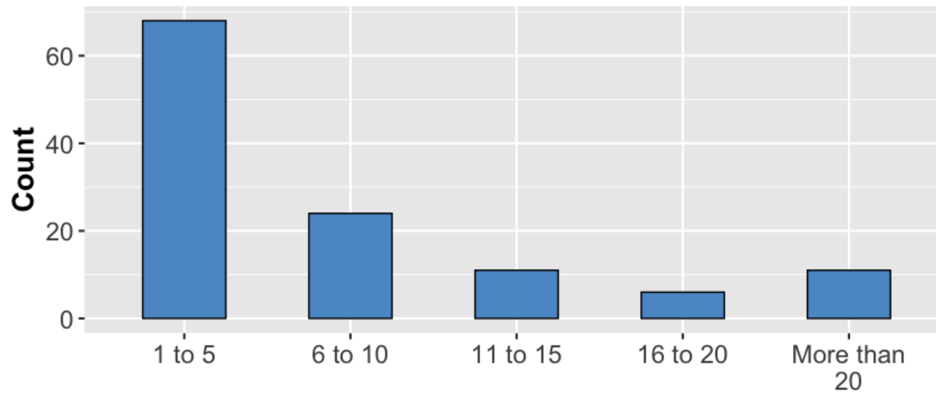

Figure 5.6: Number of developers

respondents indicated they had *low* confidence on knowledge, with none responding *very low*.

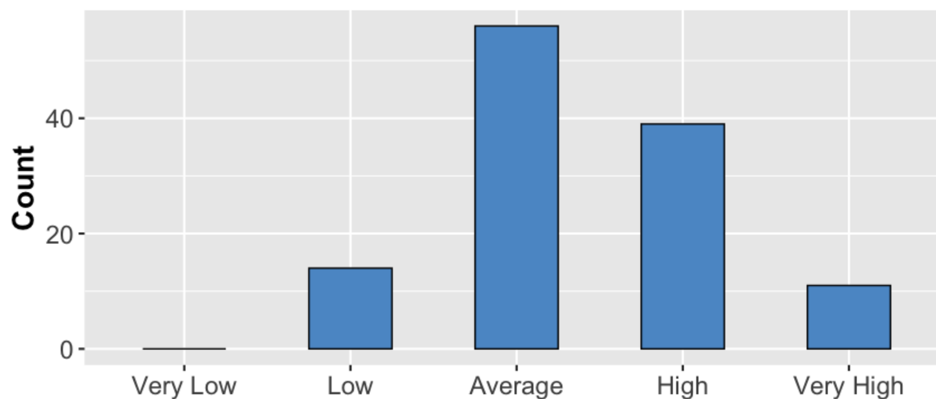

Figure 5.7: Confidence on knowledge of software testing

Next, Q7 asked respondents about their level of understanding of the testing concepts they actually used in their projects. The results in Figure 5.8 show that more than half of the respondents indicated their level of understanding was *high* or *very high*. Another 1/3 indicated their understanding was *average*. Only a small number had a *low* or *very low* level of understanding. Similarly, Q8 asked respondents about their level of understanding of the testing concepts needed for their projects, which might be different than those actually being used. The results in Figure 5.9 show a slightly different distribution than

the previous question. Most respondents still reported at least an *average* level of understanding. However, the responses shifted away from *very high* into *high* and *average*. These distributions are significantly different ($\chi^2 = 31.0068$, $p < .001$). Together, the responses to these two questions indicate that while survey respondents believed they an adequate understanding of the testing concepts used in their projects, they were less confident about the testing concepts they actually needed.

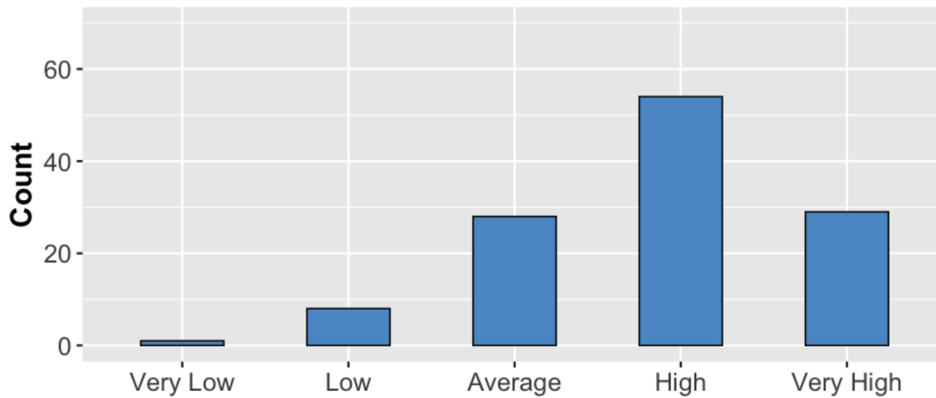

Figure 5.8: Level of understanding on the testing concepts used

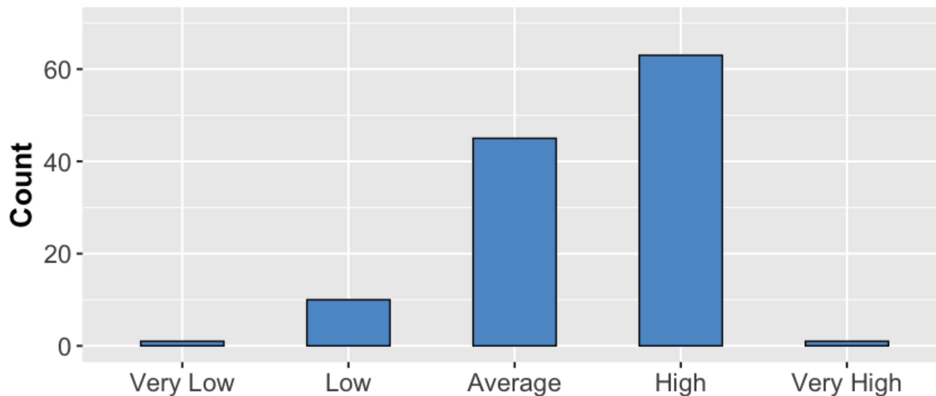

Figure 5.9: Level of understanding on the testing concepts needed

To gain a better insight into the respondents' level of knowledge, Q9 asked them to list any software testing techniques with which they were familiar. By providing a free response question rather than a set of checkboxes, this question allowed us to judge the

breadth of the respondents' testing knowledge. Across all respondents, we collected a long list of testing techniques (Appendix B.1). One interesting observation from this list is that a number of the "testing techniques" listed by the respondents are not actually testing techniques. For example, *unit testing*, *integration testing*, and *system testing* are testing "methods", not particular "testing techniques". This result suggests that while research software developers are familiar with a lot of approaches, their understanding of core software engineering terminology and their level of knowledge is likely different than that of a formally trained software engineer.

5.4.3 RQ2: How do research software developers test their software?

Given that the respondents had a reasonable understanding of testing (RQ1) and given the inherent difficulties in testing research software, the results for this question help us understand how testing occurs in practice.

Survey question Q10 asked respondents to choose which of the following testing goals (obtained from *Introduction to Software Testing* by Ammann and Offutt [3]) most closely matched their project:

- Level 0 - There is no difference between testing and debugging
- Level 1 - The purpose of testing is to show correctness
- Level 2 - The purpose of testing is to show that the software does not work
- Level 3 - The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4 - Testing is a mental discipline that helps all researchers develop higher quality software

The results (Figure 5.10) show the most common response is *Level 4*, with a large group responding *Level 1*. This result means respondents care about producing high-quality software to show correctness and doing so is a mental satisfaction to them. It

is encouraging because of having a concrete testing goal represents respondents' willingness to produce trustworthy software by employing proper testing on the projects.

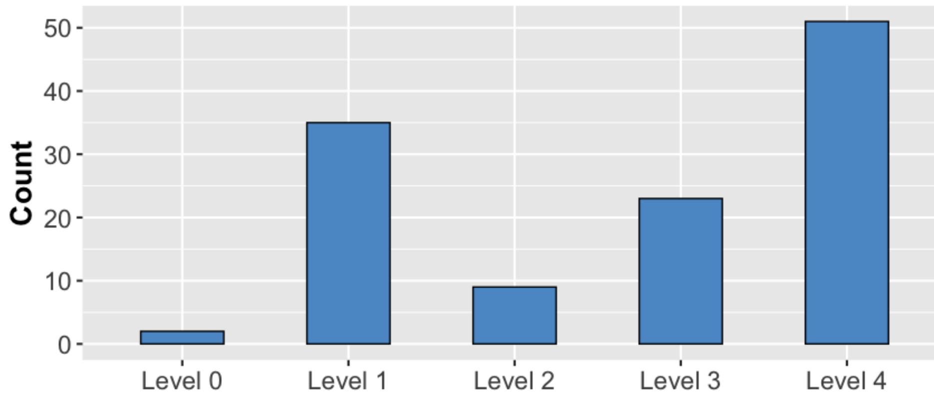

Figure 5.10: Goal of testing

The next question (Q11) asked respondents to indicate which testing methods (chosen from a list provided on the survey) their team uses. The results in Figure 5.11 show *System Testing* is the most commonly used method among the respondents. Many respondents also chose *Unit Testing* and *Integration Testing*. The use of *Acceptance Testing* and *Module Testing* is less frequent. Following on this question, Q12 asked respondents how useful they found software testing in their projects. The results (Figure 5.11) show most found testing to be useful *always* or *most of the time*. Only a few respondents indicated testing was *sometimes* or *rarely* useful, with no one indicating it was *never* useful. These results indicate that respondents found some types of testing (system, unit, and integration) to be useful most of the time.

Last, Q13 provided a list of testing techniques, along with their definitions, and asked respondents to indicate which of those techniques they actually use in their project. In addition to the techniques provided on the survey, respondents could write in other techniques they use. Table 5.1 summarizes the responses. The top portion of the table lists the testing techniques included on the survey. The bottom portion of the table contains the techniques respondents provided in the *other* section.

Table 5.1: List of Used Testing Techniques.

Used Testing Techniques (from options)			
Name	Count	Name	Count
Assertion checking	94	Error guessing	21
Performance testing	72	Fuzzing test	20
Backward compatibility testing	53	Graph coverage	17
Statement coverage	53	State transition	17
Test driven development	50	Logic coverage	11
Condition coverage	33	Decision table based testing	9
Dual coding	33	Input space partitioning	6
Branch coverage	32	Syntax-based	6
Monte carlo test	27	Using machine learning	4
Boundary value analysis	26	Equivalence partitioning	4
Metamorphic testing	26		
Others (Write-ins)			
Regression testing	4	Portability testing	1
Bit-for-bit comparison	1	Code coverage	1
Benchmarking	1	Scaling test	1

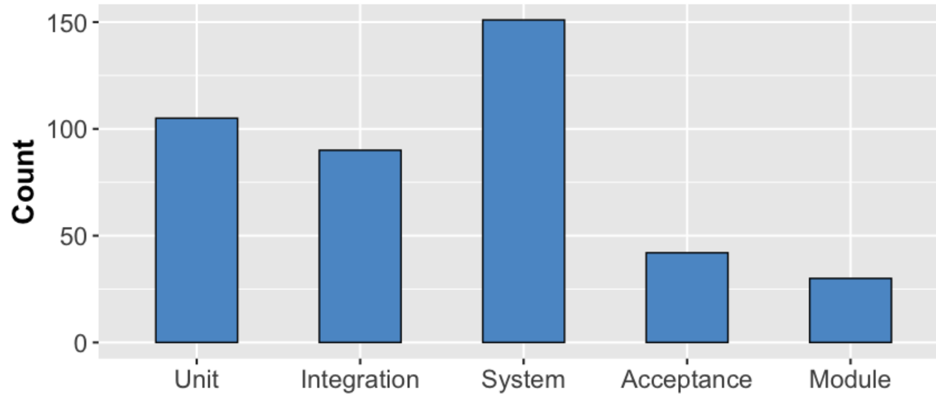

Figure 5.11: Testing methods used

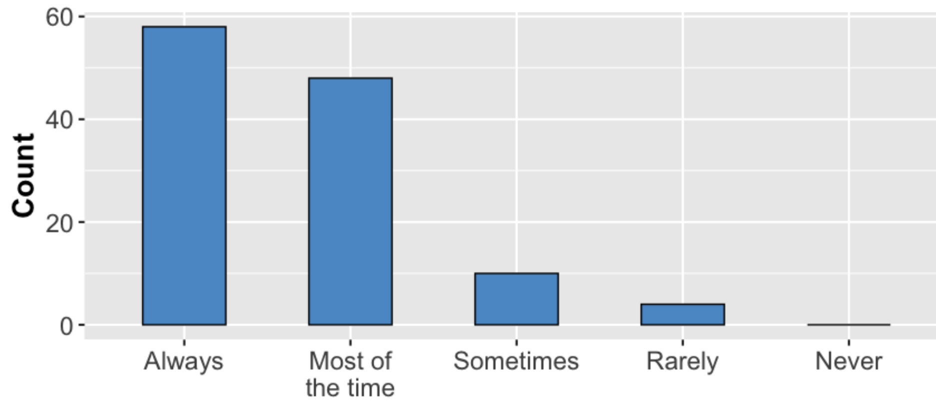

Figure 5.12: Usefulness of testing

This result suggests that research software developers use a wide variety of testing techniques in their projects. This result is also consistent with the results in Figure 5.7 showing that respondents have an adequate level of knowledge and Figure 5.8 showing they have good understanding of the testing concepts used in their projects.

5.4.4 RQ3: Why is testing research software difficult?

To make any progress in overcoming the difficulties with testing research software, it is important to understand the specific reasons why testing is difficult. This research question helps identify specific challenges and barriers. Survey question Q14 asked respondents to rate the complexity of testing their projects. According to the results (Figure 5.13), the

distribution of responses is skewed towards the lower complexity end of the scale, with the peak at *Moderately Complex*. This result means that while there is some level of complexity in testing, most respondents do not find the complexity to be to high.

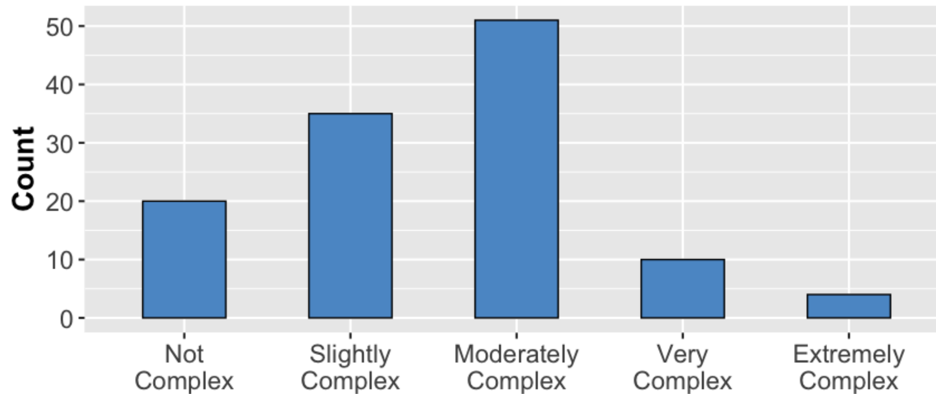

Figure 5.13: Complexity of testing research software

To gain a deeper understanding of the difficulties with testing research software, Q15 asked respondents to explain any barriers or challenges they face with testing their software. This question was open-ended and allowed respondents to write their own thoughts. Our qualitative analysis of these free-response answers resulted in the 12 high-level categories challenges shown in Figure 5.14. The following text goes through each high-level challenge to explain what it means.

The most commonly mentioned class of challenges is **test case design**. One respondent described the challenge as having difficulty *"[e]ngineering good test cases and making sure that all equivalence classes of test cases are covered"*. Another respondent indicated that *"[a]pplication cases are usually too big/expensive to test, so breaking them down for meaningful system tests is a challenge"*.

The second most commonly mentioned challenge is the **lack of resources**, which includes, as one respondent said, *"[l]ack of funding, people, and calendar time"*. Another respondent commented that *"[t]he amount of time needed to write tests gives them a bad reputation with developers who aren't convinced they are necessary"*. Another respondent

described the problem as *"[l]imited amount of time is "allowed"/"allocated" for writing tests and setting up testing environments"*.

The third most commonly mentioned challenge is **external dependencies**. The following comment summarizes the thoughts of many respondents: *"Tests should execute external proprietary software which is unavailable on services like TravisCI"*.

The fourth most commonly mentioned class of challenges is **lack of knowledge**. One respondent described the problem as *"..lack of testing knowledge – collectively, as a team, we've probably heard of all of the possible ways of testing mentioned in the questions above, but several of these categories are not well understood by the team (or any individual within in).."*.

The fifth most commonly mentioned class of challenges is **slow**. One respondent described this problem as *"[t]ests take a long time to run, slows down continuous integration"*.

The sixth most commonly mentioned class of challenges is **culture**. One respondent explained the challenge to testing their as *"[m]ostly cultural, convincing my team mates that this is important to the sustainability of the code base"*. Moreover, the research software culture is *"[a] culture that doesnt value test development"*.

The seventh most commonly mentioned class of challenges is that testing **affects continuous integration**. As a summary of this class of responses, one respondent indicated *"[t]esting across multiple machines regularly is a challenge, due to the continuous integration tests running on a single machine"*.

The eighth most commonly mentioned class of challenges comes from the fact that **comparing results with reality** makes testing very difficult. As a specific example, one respondent said *"[s]ince we're developing a computational fluid dynamics code (it's an ocean model), the most difficult part was testing that the model produces the physically correct output"*.

The ninth most commonly mentioned challenge is a result of the **codebase** itself. One

respondent describes the problem as *"[t]he code has not been designed in a very modular way, so unit testing is not easy to implement"*

The tenth most commonly mentioned challenge is **legacy code**. Sometimes in research software development there are *"[l]arge amounts of legacy code that were not developed with testing in mind"*.

The last class of challenges is **cost**. As one respondent indicated, the *"high cost in maintaining tests, expensive/slow to run full test suite"* makes the testing research software difficult.

In addition to these high-level classes, there were a number of **other** challenges, including environmental changes, testing graphics production, and dependencies on programming languages and database systems.

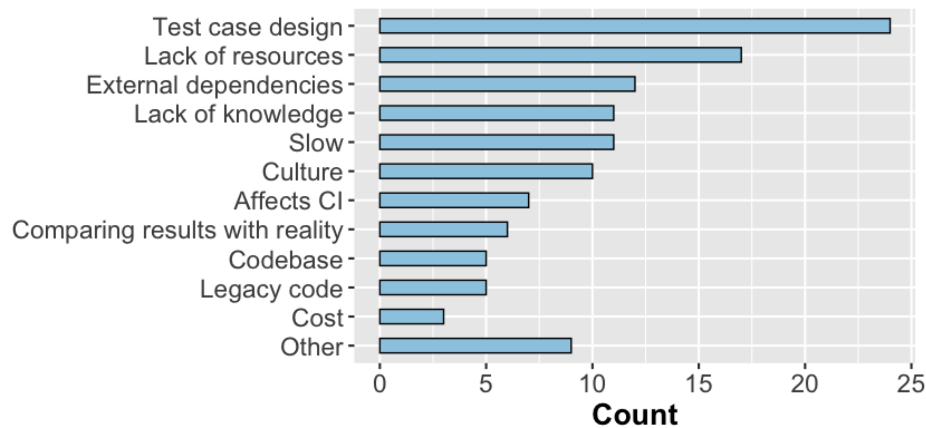

Figure 5.14: Challenges in testing research software

5.4.5 RQ4: Is it possible to adapt existing testing techniques to support the testing of research software?

In this case, "existing testing methods" refer to the testing methods currently used in Commercial/IT software development. These methods include unit testing, integration testing, system testing, acceptance testing, and module testing. Gaining a better understanding of how these methods apply to the testing of research software will help research software developers have more confidence in the methods they can use and reduce

the need to discover this information on their own.

We began with two survey questions about the frequency with which respondents use Commercial/IT testing methods as a team (Q16) and individually (Q17). As Figures 5.15 and 5.16 show most teams and individuals apply these methods at least *sometimes*. There is no significant difference between the distribution of responses. Beyond whether the respondents use the Commercial/IT testing methods, Q18 ask the level of value they see personally in using these methods. The distribution of results in Figure 5.17 show that the respondents generally saw value in using such methods, with more than half answering *high* or *very high*.

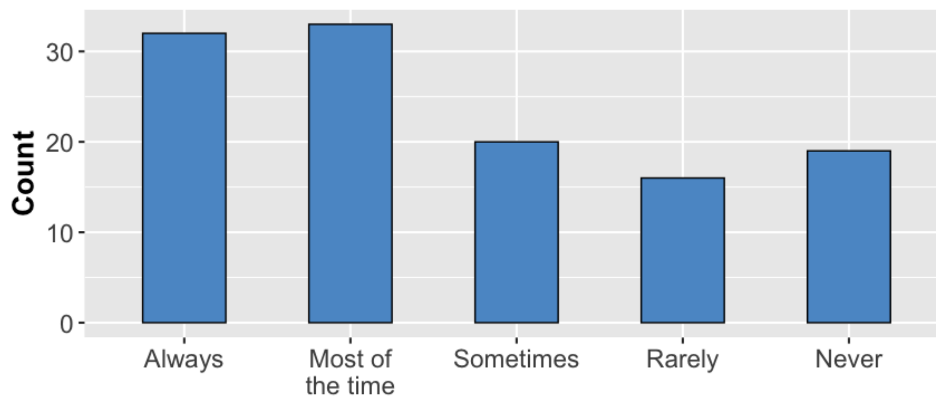

Figure 5.15: Applying Commercial/IT testing methods by team

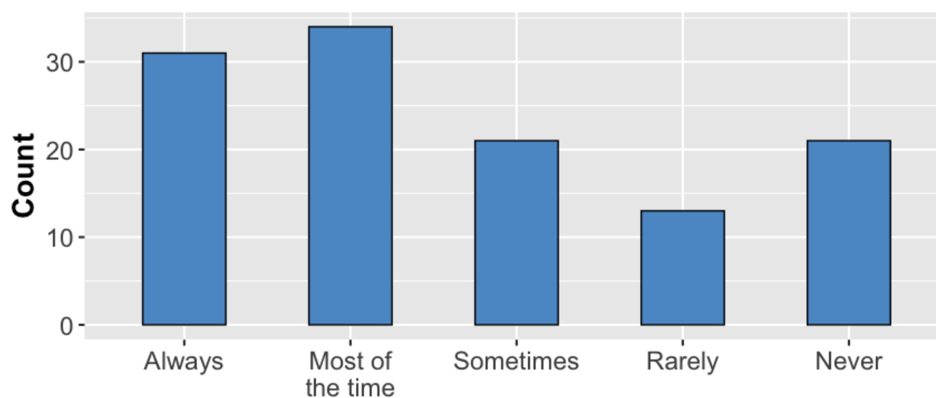

Figure 5.16: Applying Commercial/IT testing methods personally

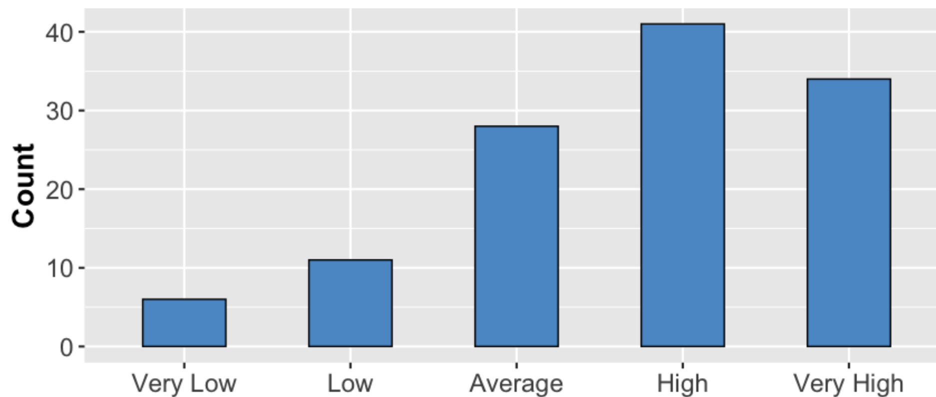

Figure 5.17: Value seen in using Commercial/IT testing methods

To gain insight into where the Commercial/IT methods cause problems, Q19 asked respondents to explain any challenges they faced in an open-ended manner. Our qualitative analysis of the results identified the nine high-level categories of challenges shown in Figure 5.18. Because we discuss some of these challenges in detail in response to other questions, we only highlight a subset here.

Overwhelmingly, the most common challenge respondents reported was that the methods were **not useful**. This challenge arises because *"[r]esearch software is typically not production software"* and *"[s]ome commercial tools do not account for issues with numerical tolerances. Legacy codes are hard to get under test"*. Another respondent indicated it is *"Difficult to adapt [Commercial/IT testing methods] to the development of scientific software because of numerical errors and often not knowing the expected output"*.

Similar to the overall difficulties with testing research software, the second most common challenge to using Commercial/IT testing methods is **lack of resources**. One respondent explained the challenge as *"[l]ack of expertise, schedule demands, lack of R&D (i.e. exploratory) s/w development oriented tools"*.

The third most common response is the **mindset** of the research software developers. One respondent summarized the problem as *"[d]evelopers often feel like any time not spent developing code for the core software is not real work, thus time spent writing tests is less*

enjoyable". In addition, the problem is *"[c]ultural – convincing people that it is beneficial, necessary, and worth their time and not insulting to their work"* There are also challenges with mindset that originate external to the team, such as *"[o]ur funding agencies are generally not aware of their importance, and effort spent on testing is effort not spent writing/publishing papers"*. Finally, there is a *"[l]ack of familiarity with the ensemble of testing patterns and goals available"*.

Moreover, there is a lack of knowledge, need for infrastructure support, expensive or difficult to use, and runtime restrictions that make research software difficult to adapt Commercial/IT testing methods.

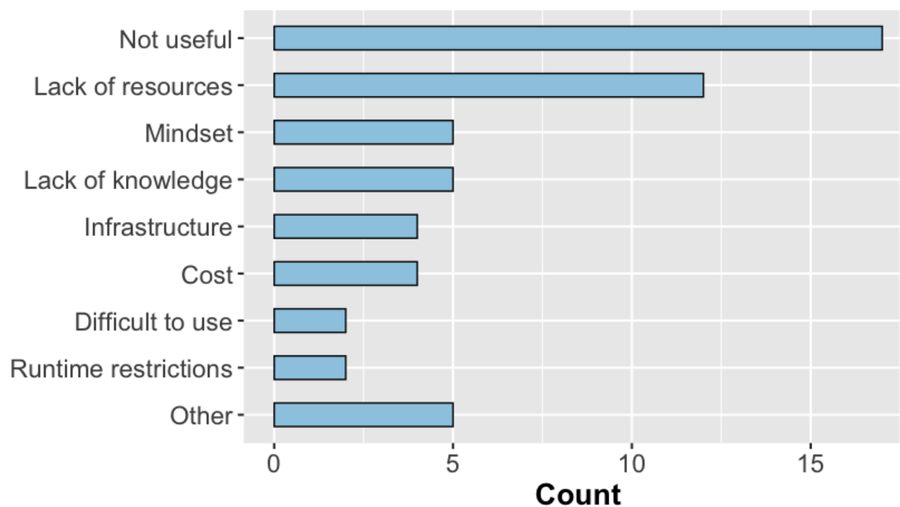

Figure 5.18: Challenges to adapt Commercial/IT testing methods

In addition to challenges with adapting Commercial/IT methods, there are cases where those methods are just not applicable. Survey question Q20 asked respondents to explain any challenges that could not be met by Commercial/IT testing methods. The qualitative analysis produced five high-level challenges (Figure 5.19).

The most common challenge, by a large margin, was that the methods do not meet the **specific needs** of research software. These specific needs can include *"[v]isualization [, i]mage processing & analysis [, f]luid flow simulation [, s]ituations where there is no analytic solution or known correct answer [, i].e. no oracle"*. A unique challenge of

research software is that *"[v]alidation requires domain expertise which is sometimes difficult to express in the commercial methods"*. Another situation that is common in research software is whether the results are meaningful, as a respondent stated *"IT methods are good for preventing errors / seg faults, but not good at catching if numbers are no longer meaningful (eg demand curves have inverted) or triaging to determine"*.

The respondents also mentioned **lack of expertise**, **slow** to execute the test, **continuous integration issues**, and **benchmarks** as some of the challenges that could not met by Commercial/IT testing methods.

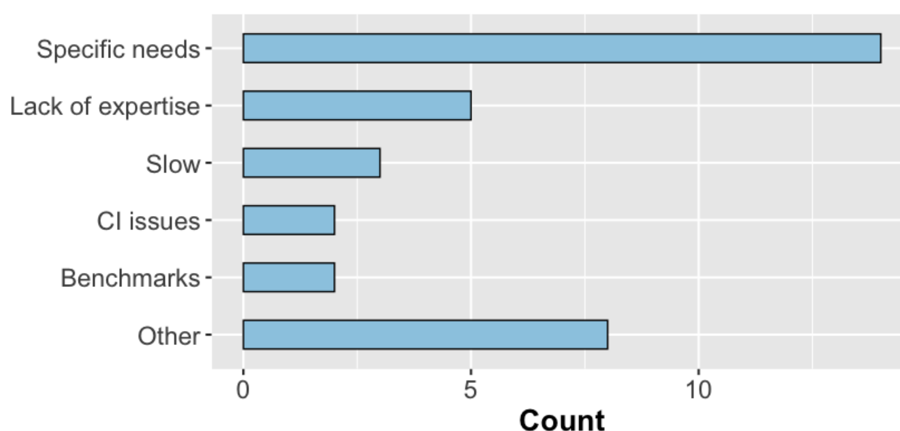

Figure 5.19: Challenges could not met by Commercial/IT testing methods

5.4.6 RQ5: What improvements to the testing process do research software developers need?

Finally, to help provide more concrete guidance to the research software community, Q21 asked respondents to describe how to improve the testing process. The qualitative analysis produced ten high-level categories of improvements (Figure 5.20).

The most commonly mentioned improvement was proper **training**. Some of the specific suggestions provided by respondents include: *"[t]each it to grad students as an essential part of writing software"*, *"[m]ore training, earlier on, in scientists' careers"*, *"[s]eminars for practitioners, classes on scientific software engineering"*, and *"[e]ducate scientists about the benefits of testing software"*.

Then the second most commonly mentioned improvement was **more tests**. This improvement focuses more on developer behavior as suggested by *"[w]rite more unit tests, preferably using a dedicated testing framework as pFunit"* and by *"[i]ncreased use of customized fuzz testing covering more input features"*.

The third most commonly mentioned improvement was **infrastructure**. One respondent requested *"[a] public service for testing that (1) is freely available for open source, (2) with many-tier (detailed, incremental) pricing structure for more machine time if needed, with (3) a sophisticated testing dashboard, similar to that of TeamCity"*. Another respondents described the need as *"[o]ur group needs more developers. This is highest priority. But, just as high: we need support from the academic infrastructure providers to be able to run tests before our users hit the resources"*. Another respondent wanted a *"[simplified] infrastructure for creating new tests"*.

Following from that response, the fourth most commonly mentioned improvement was **automation**. Specifically respondents wanted automation for *"setting tests and analysis of results"* and *"simpler methods for enabling/using tools"*.

Tied for the fourth most commonly mentioned improvement was **continuous integration**. These systems need improvement because *"[o]ur CI / testing system is very frail, and tends to break when there are system updates, requiring supervision and triage, and then there isn't a easy way to rerun tests on PRs that came in during the down time"*.

Also tied for the fourth most commonly mentioned improvement was the changing the **culture** of testing in the research software developer community. A respondent explicitly said *"Changing the culture. Ensuring any pull requests that come in have adequate code coverage. Incentivizing test development efforts. Educating the developers on the importance of software testing and how best to go about it. Making sure when bugs are getting fixed that the fix isnt accepted until theres a new unit test in place to cover the bug"*. Similarily, there is a need to *"[c]onvince other developers that are a part of the same project to adapt to these practices"*.

Also tied for the fourth most commonly mentioned improvement was the need to **improve code quality**. This improvement has to happen among the developers themselves, as one response suggested to *"[h]ave HPC researchers, scientists, and engineers write better code. While improvements can certainly be made in testing, the vast majority of the problem currently is the state of the code written. You can't really test a 1000-line main() - you need structure, seams, and modularity in whatever language or paradigm you use"*. Another respondent suggested that developers need to *"Simplify[] code, reduce redundant tests, design better input variables, study other individual components and other possible problems with different architectures or compilers, etc"*.

Also tied for the fourth most commonly mentioned improvement is the need for proper **acknowledgement** to help motivate the research software developers to improve their testing. There is a need for *"[f]unding agencies and application users understanding the benefits of testing and support it enthusiastically"* and a need for *"better incentive for others to contribute tests, period"*.

Moreover, initiatives to **make simpler** and provide adequate **resources** could potentially improve the testing process in research software development.

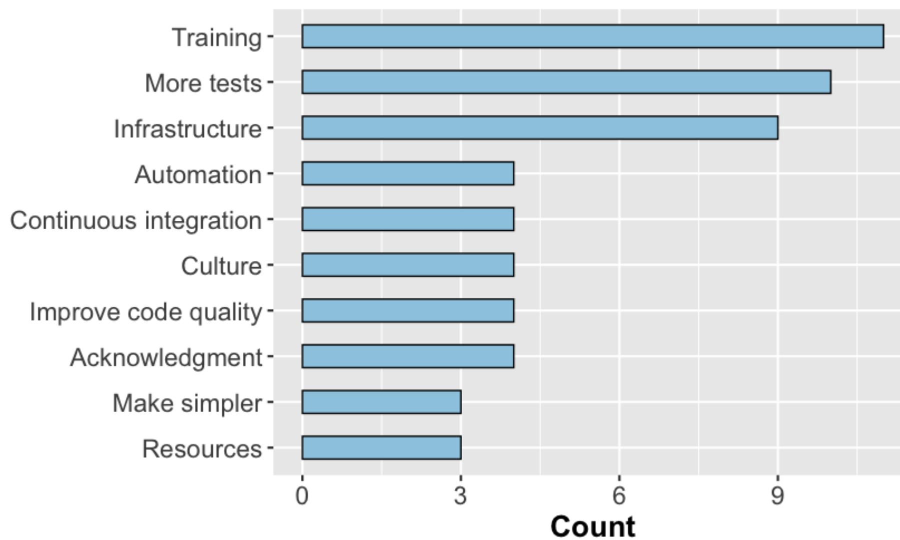

Figure 5.20: Improvement of the testing methods

5.5 Discussion

Based on the detailed results in the previous section, this section discusses key insights gained relative to each of the overall research questions.

5.5.1 RQ1 - Knowledge

In general, research software developers are confident about their knowledge of software testing. The survey respondents think they have at least an average level of understanding of the testing concept used and needed in their project. Our findings are somewhat inconsistent with the previous literature (Section 5.2) indicating that research software developers have little or no knowledge of software engineering. One reason for our different results may be a growing awareness of software engineering among the research software community. This result is encouraging because as research software developers become more knowledgeable about software engineering practices, they will tend to produce higher-quality software.

When we asked respondents to report the testing techniques with which they are familiar, the responses were a mix of testing techniques and testing methods. Therefore, while respondents may report their knowledge is adequate, it is clear from the results that there is still a lack of understanding of common software engineering terminology. Therefore, there is still a need for more formal training to increase the level of software engineering knowledge of research software developers.

5.5.2 RQ2 - Practices

In terms of performing software testing, research software developers have a clear goal. Many of the respondents viewed testing as a mental discipline that makes them confident in producing trustworthy results. Again, this result is somewhat inconsistent with prior research (Section 5.2) indicating that research software developers have no written quality goal. Conversely, another set of respondents viewed the goal of testing as showing the correctness of the software, which evidences a much lower maturity level. These

inconsistent results show that there is still a large disparity in knowledge among the research software developer community.

The literature did not provide much evidence that research software developers use different testing techniques. However, the results in Figure 5.11 show that research software developers do, in fact, use a wide variety of testing methods. The respondents also reported using a wide variety of software testing techniques. Most of the respondents indicated that testing was useful at least most of the time.

5.5.3 RQ3 - Difficulties

In terms of challenges and barriers to testing research software, our results are consistent with the literature. The respondents indicated that there is some level of complexity in testing their projects. The survey also identified a number of challenges for testing research software. Test case design, lack of resources, and external dependencies are the most common challenges. Lack of knowledge, test execution slowing down the development process, the lack of a proper testing culture all make testing difficult. Even though our results are consistent with the literature, we were able to contribute a list of concrete and current challenges taken directly from the experiences of research software developers.

5.5.4 RQ4 - Adapting existing testing methods

In terms of applying existing Commercial/IT testing methods for testing research software projects, our results are inconsistent with the literature. Most of the respondents indicate that they apply Commercial/IT testing methods at least sometimes both personally and as a team. They see at least an average level of value in applying Commercial/IT methods, while more than half of the respondents see high or very high value. While this result contradicts the published literature, it is encouraging. Applying available testing methods can be beneficial to many research software developers.

Even with the positive view of Commercial/IT testing techniques, there are still some challenges to adapting those techniques for use in research software. In many cases,

Commercial/IT testing techniques are often not useful. Respondents identified specific needs of research software that could not be met by the existing testing methods. In spite of the challenges to adapting these testing methods for use in research software, this result is encouraging because it indicates that many research software developers are trying to apply these methods.

5.5.5 RQ5 - Improvement

We identified a large number of potential improvements to the testing process for research software. These results can serve as guidance to software teams who want to incorporate good practices as well as to testing researchers who want to provide more appropriate techniques for research software. There is a need for formal training to help research software developers better understand testing concepts and the importance of testing their projects. In addition, there is a need for more tests and better infrastructure. Moreover, changing the culture of research software to one that embraces testing, improved code quality, and proper acknowledgment for testing effort will help improve the overall testing process.

5.6 Threats

This section describes the threats to validity of the study.

5.6.1 Internal Threats

The primary threat to internal validity is whether participants understood the software engineering concepts in the same way we intended them. If the respondents did not understand the concepts or had different definitions, then the results we obtained would be less reliable. Because the members of the target survey population are not traditional software engineers, it is possible that they lacked the necessary knowledge to properly answer the questions. However, the results of the open-ended questions showed that respondents used many appropriate software engineering terms to describe their

practices. Therefore, because the respondents did have adequate software engineering knowledge to participate in this study, this threat is minimal.

5.6.2 External Threats

If the survey respondents are not representative of the larger population of research software developers, the results are less generalizable. To reduce this threat, we recruited participants from different countries and different projects. While it is clear that the participants are all research software developers, some of the responses suggest that they may be more interested in software testing than the average research software developers. In addition, they took time to answer a survey about testing. Therefore, the responses may be biased towards those developers who are already predisposed towards the use of testing. Therefore, the results must be considered in this light. We would expect that the results from the larger research software community to be no more positive than the ones we received from this sample.

5.6.3 Construct Threats

The primary construct validity threat is the participants may have misunderstood the questions. We took great care in writing the survey questions and verified them by expert research software developers and software engineering researchers. In addition, we provided enough definitions to the respondents without biasing them so they could used their own judgment to respond.

5.7 Conclusion

Testing is a popular and essential software quality practice for building high-quality and trustworthy software. Because research software often supports critical situations and produces evidence for research publications, it is important for researchers to use appropriate testing approaches to complement their development methods. To gain insight into the practice of testing for research software, we conducted a survey of practicing

research software developers. The the results from this survey help other research software developers understand how their peers testing their software. The results also help software engineering researchers and tool developers understand the specific needs of research software developers and provide suggestions for improvements to testing methods and tools that would help the research software community.

In this paper, we report insights about testing research software gained from 120 responses to a survey of research software developers. The paper discussed the overall knowledge of software testing among research software developers, current practices of testing, difficulties to test research software, challenges to adapt the existing testing methods, and potential ways of improving the testing process.

Research software developers are somewhat confident in their knowledge of testing but there is a confusion of differences between testing techniques and methods. Overall, research software developers have clear testing goals and use many types of testing techniques. However, there are many challenges and barriers involved in testing research software. In addition, use of Commercial/IT testing techniques is not easy or sufficient to be useful in all cases. Proper training and creating a culture of testing could potentially address many of the difficulties research software developers face and improve the testing process to support production of high-quality research software.

In the future, we plan to work closely with research software teams to identify and address the specific challenges they face when testing research software. By gathering these real experiences, we will be able to develop best practices around testing that will be of value to the research software community.

CHAPTER 6

CONCLUSION

This chapter provides a summary of this dissertation followed by the contributions, recommendations, and future works.

6.1 Summary

I used Empirical Software Engineering methods to investigate different software quality activities in research software development and reported the results in this dissertation. As a unique field of study, software engineering researchers have not deeply studied software quality assurance in research software. Regarding the increased development and use of research software along with the importance of producing high-quality research software, I chose four key software quality practices: software metrics, software development process, peer code review, and software testing to be included in this dissertation. I described all of these practices in four distinct articles. I collected data through surveys, interviews, and directly working with research software developers. I gathered information regarding research software developers' knowledge and use of these four quality practices, current practices in the research software developer community, challenges and barriers they face to employ these practices, and potential ways to improve the practices in their project. In the next sections, I describe the contributions of this dissertation and recommendations that will help the research software developer community in practicing and adopting software quality activities in their project and build high-quality software.

6.2 Contributions

I provide a brief overview of each of the four software quality activities in this section.

6.2.1 Metrics

Overall, research software developers have low knowledge of metrics. Due to this low knowledge of metrics, they never found metrics useful in their projects. But they have a wide variety of interests on different types of metrics. Our result shows that, besides the available code, process, testing, and general quality metrics, research software developers are also interested in performance metrics and recognition metrics. These new categories are often of interest to research software developers working in high-performance computing environments. Recognition is particularly timely as research software developers are increasingly interested in being recognized and receiving proper credit for developing research artifacts such as software, tools, and libraries.

One important aspect of metrics is to measure individual or team productivity. But the frequency of using metrics for evaluation is very low in research software development. In terms of code complexity problems, research software developers found code complexity is at least sometimes a problem. But they never use code complexity metrics and that is why they think code complexity metrics are never useful.

The demographics of the respondents do affect their perception and use of metrics. Our analysis of the influence of project size shows that respondents from smaller projects appear to have less overall knowledge of metrics and see metrics as less useful than respondents from larger teams. This result could be due to the typically smaller amount of resources smaller teams have to devote to metrics. The analysis also shows that respondents from different project stages differ in terms of knowledge and perceived usefulness of metrics.

Software metrics are very useful in quantifying characteristics of the software, software developer, and the development team. It helps in producing high-quality software and

measures the productivity of the team. Research software developers should choose from many available metrics and utilize them in their projects. Software engineering researchers should develop new metrics according to the necessity of research software.

6.2.2 Process

Research software developers have an overall tendency of following a defined software development process at least sometimes. It is also not surprising that they see moderate value in following a process. An encouraging result is the identified relationship between a respondent's perceived value of using a defined software process and their likelihood of actually using a defined software process. This result suggests that one way to increase the level of software process in research software is to help research software developers understand the value of following software process. An interesting finding is that the use of agile methods includes the use of lighter weight methodologies, including Use-Case Methodology and Rapid Application Development in their work. That means an agile-like approaches may be a better fit for research software development.

A somewhat discouraging finding is that although respondents see value in process, they rarely use process metrics to evaluate the success of software process. In the absence of process metrics, it is difficult to imagine how quality control can be ensured, because process effectiveness is typically measured at every stage to reduce defects in production.

Similar to the demographic influence in using metrics, larger teams followed a defined process more often than small teams and see more value in using a defined software development process. This result makes sense because the larger the team gets, there is more need for a defined process to manage the development.

Overall, the results of using a defined software development process were encouraging, but research software developers should try to incorporate process metrics into their development as well.

6.2.3 Code Review

Peer code review helps research software developers to produce high-quality software. Overall, research software developers typically follow an informal review process. Most of the respondents indicated they initiate code review with their peers through pull-request on GitHub, BitBucket, or GitLab. Code reviews help research software developers identify many problems in the code. Most of the respondents identified problems related to software quality and code mistakes.

Research software developers have overall positive experiences regarding code review. They found code review as a good way of knowledge sharing. They were able to improve the code quality through good feedback and problem identification. Their major negative experiences are the process takes too long and misunderstanding the criticism. There is a lack of people who have both domain knowledge and coding knowledge to take part in the review process. Moreover, having time to do the code review is the biggest barrier they face in the code review process.

In order to improve the code review process and make it more effective, research software development teams should make it a habit by formalizing the process and using appropriate tooling. There is a need for more people to participate in the code review process, so, providing proper incentives and training could potentially increase participation. Research software developers should invest more time on code review and make the process faster by providing quick feedback. Improving the code review process can help eliminate many problems that are not addressed by testing.

6.2.4 Testing

In terms of testing, research software development teams have clear goals. Research software developers tend to use system testing, but they use other types of tests as well. It is not surprising that they found testing very useful. But there are a lot of challenges to testing research software. Test case design and lack of resources are the two major challenges they face. There are many kinds of dependencies on testing research software.

Sometimes, research software developers develop software to compare with the reality which is difficult to test. Moreover, there is not a culture of testing in most of the research software development teams. The existing testing methods are not often useful because of the specific needs of the research software.

There is a clear benefit of using available statistical tests to test research software. Our testing framework developed for testing the *ParSPlice* project does not test the actual output of the project but provides the confidence in producing correct output to its developers and users. The lessons learned from this case study can be valuable to the larger research software community because, like *ParSplice*, many research software projects have stochastic behavior that produces non-deterministic results. The approach we followed to develop the test framework can be a model for other research software projects.

In order to improve the testing process in research software, proper training is the foremost thing to take into account. There are needs for more types of tests and proper infrastructure support. Test automation can reduce many burdens on the development point of view. Moreover, proper acknowledgment along with improving the code quality and making a culture of test can potentially improve the testing process significantly. Research software developers should improve their testing process to produce high-quality software and trustworthy results.

6.3 Recommendations

Based on the results discussed in this dissertation, I provide a set of recommendations that research software developers can incorporate in their projects to produce high-quality research software.

6.3.1 Metrics and Process

- Providing resources and training to increase the knowledge of metrics in the research software developer community.

- Installing proper metrics program in the projects to see the benefit and letting other people know of the benefits.
- Utilizing the available software metrics in the projects and develop new metrics according to the necessity, for example, performance and recognition.
- Using metrics to evaluate the productivity of developers and teams.
- Using code complexity metrics to reduce complexity in the code.
- Following a defined software development process each time to develop a software.
- Using process metrics for evaluation of effectiveness.

6.3.2 Code Review

- Making the code review process more formal with a structured guideline for each step of the process.
- Trying to ensure at least one science review and one technical review.
- Including automatic tools in your code review process and train your peer reviewers the best practices to use the tool.
- Encouraging more people to participate in the review process and allocate some time to do the review.
- Providing incentives or some kind of rewards to the reviewers to participate in the code review.
- Allocating a good amount of time in the development process to perform code review.
- Providing faster response to any incoming review request.
- Training reviewers on how to phrase good feedback.

- Training developers to forget their egos and accept comments from the reviewers to improve their code.
- Making the overall code review process faster.
- Providing necessary support from the administrative level that encourages people to participate in the code review process.

6.3.3 Testing

- Providing enough training on software testing to all kinds of research software developers ranging from graduate students to experienced researchers.
- Incorporating more tests that can solve specific needs of the research software.
- Providing infrastructure support, for example, a public service for testing including many-tier pricing structure for machine time and a sophisticated testing dashboard.
- Providing automation for setting tests and analysis of the results.
- Improving continuous integration system to facilitate a better way of testing, especially, the incoming tests during down time.
- Making a culture of testing in the team and encourage others by sharing the benefits from the experience of testing.
- Improve the quality of the code so that developers can write tests easily.
- Providing proper acknowledgement to developers for contributions in testing.
- Making the testing process simpler so that it is easy to adopt in the project.
- Providing enough resources to developers so that they can utilize the resources to develop test suites.

6.4 Future work

As a software engineering researcher for research software, I always felt the necessity of machine learning techniques to be integrated with software engineering techniques. Because of the stochastic characteristics research software poses, developers certainly require machine learning techniques to utilize the available software quality practices. For example, Metamorphic testing is a potentially useful technique for testing research software, but it requires both a good understanding of machine learning techniques and knowledge of the underlying science. While the commercial/IT software industry has begun including machine learning techniques in their development process, it is time for research software to do so.

In the future, I plan to continue collaborating with the National Labs and the DOE IDEAS-ECP team to develop software quality techniques and tools for research software. I am particularly interested in working on software quality, especially testing, in research software and integrating machine learning techniques into the testing process. Another future area of research is to apply traditional software engineering concepts, such as, metrics and process, on actual research software projects and determine the modifications necessary for these projects. Building on the findings of this dissertation, I would like to work more closely with research software projects to better understand how these results apply in practice. By interacting directly with research software developers as they write software, I can conduct direct observational studies to gather information in real-time rather than relying on the type of retrospective reports that we obtained from our interviews and surveys. Directly observational studies bypass the problems of self-report measures and there can be no covering up or no false reports. Direct observation also allows people to discuss real, indisputable actions as they occur. By gathering these real experiences, I will be able to develop best practices around the software quality practices that will be of value to the research software community.

As part of the BSSw (better scientific software) fellowship, I will develop a specialized

tutorial on testing research software. I will start the tutorial with background information and the usefulness of using automatic testing techniques in research software development. During this portion of the tutorial, I will present challenges, potential solutions, and unsolved problems faced while testing research software from articles 3 and 4 of this dissertation. The second part of the tutorial will be hands-on. I will cover different testing techniques, such as input space partitioning, test-driven development along with some testing techniques specially designed for scientific software such as metamorphic testing and run-time assertion. I will conclude the tutorial with a large group discussion to gather input from the participants about which approaches are more suitable for their individual projects. I will present the tutorial at relevant conferences and national labs when possible. I would like to present a BSSw webinar once the tutorial is more mature. I also plan to write a few BSSw blog posts from the results discussed in this dissertation.

Besides the quality activities discussed in this dissertation, future investigation on requirements analysis, design methods, and refactoring could be helpful in order to help research software developers in efficiently develop and maintain software. Requirements analysis is important because it contains the behavior, attributes, and properties of the future system. Following proper design methods allow research software teams to be efficient, transparent and focused on creating the best product possible. Refactoring helps with easier maintainability and facilitate reuse of the software. These efforts could also lead research software teams to value the life-cycle of software development in a manner similar to how more traditional software teams value.

My research will help to address some of the most important problems that require approaches that are fundamentally multidisciplinary, for example, weather forecasting, climate change, and cancer research. These multidisciplinary problems require skills of computer science and understanding of another discipline. I will try to bridge the gap between the researchers from multiple disciplines and computer science by bringing valuable software quality practices into the research software community.

REFERENCES

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, May 1989.

[2] K. S. Ackroyd, S. H. Kinder, G. R. Mant, M. C. Miller, C. A. Ramsdale, and P. C. Stephenson. Scientific software development at a research facility. *IEEE Software*, 25(4):44–51, July 2008.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 2nd edition, 2016.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721. IEEE Press, 2013.

[5] Robert Baxter, Neil Chue Hong, Dirk Gorissen, James Hetherington, and Ilian Todorov. The research software engineer. In *The Research Software Engineer*, 9 2012. Digital Research 2012 ; Conference date: 10-09-2012 Through 12-09-2012.

[6] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211. ACM, 2014.

[7] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142, Oct 2013.

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

[9] A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 146–156, May 2015.

[10] J. Carver, D. Heaton, L. Hochstein, and R. Bartlett. Self-perceptions about software engineering: A survey of scientists and engineers. *Computing in Science Engineering*, 15(1):7–11, Jan 2013.

[11] J. Carver, D. Heaton, L. Hochstein, and R. Bartlett. Self-perceptions about software engineering: A survey of scientists and engineers. *Computing in Science Engineering*, 15(1):7–11, Jan 2013.

[12] J. C. Carver. Software engineering for science. *Computing in Science Engineering*, 18(2):4–5, Mar 2016.

[13] J. Cates, D. Weinstein, and M. Davis. The center for integrative biomedical computing: advancing biomedical science with open source. In *3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006.*, pages 694–697, April 2006.

[14] W.K. Chan, S.C. Cheung, Jeffrey C.F. Ho, and T.H. Tse. Pat: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. *Journal of Systems and Software*, 82(3):422 – 434, 2009.

[15] T. Y. Chen, T. H. Tse, and Zhiquan Zhou. Fault-based testing in the absence of an oracle. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pages 172–178, Oct 2001.

[16] T. Clune and R. Rood. Software testing and verification in climate model development. *IEEE Software*, 28(6):49–55, Nov 2011.

[17] John B. Drake, Philip W. Jones, and Jr. George R. Carr. Overview of the software design of the community climate system model. *The International Journal of High Performance Computing Applications*, 19(3):177–186, 2005.

[18] A. Dubey, K. Antypas, A. Calder, B. Fryxell, D. Lamb, P. Ricker, L. Reid, K. Riley, R. Rosner, A. Siegel, F. Timmes, N. Vladimirova, and K. Weide. The software development process of flash, a multiphysics simulation code. In *2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, pages 1–8, May 2013.

[19] S. M. Easterbrook and T. C. Johns. Engineering the software for understanding climate change. *Computing in Science Engineering*, 11(6):65–74, Nov 2009.

[20] Steve M. Easterbrook. Climate change: A grand software challenge. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 99–104. ACM, 2010.

[21] S. L. Eddins. Automated software testing for matlab. *Computing in Science Engineering*, 11(6):48–55, Nov 2009.

[22] N. U. Eisty, G. K. Thiruvathukal, and J. C. Carver. A survey of software metric use in research software development. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 212–222, Oct 2018.

[23] Nasir U. Eisty, George K. Thiruvathukal, and Jeffrey C. Carver. Use of software process in research software development: A survey. In *Proceedings of the Evaluation and Assessment on Software Engineering*, EASE '19, pages 276–282. ACM, 2019.

[24] P. E. Farrell, M. D. Piggott, G. J. Gorman, D. A. Ham, C. R. Wilson, and T. M. Bond. Automated continuous verification for numerical simulation. *Geoscientific Model Development*, 4(2):435–449, 2011.

[25] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development. CRC Press, Boca Raton, FL, 3rd edition, October 2014.

[26] Kambiz Frounchi, Lionel C. Briand, Leo Grady, Yvan Labiche, and Rajesh Subramanyan. Automating image segmentation verification and validation by learning test oracles. *Inf. Softw. Technol.*, 53(12):1337–1348, December 2011.

[27] Carole Goble, James Howison, Claude Kirchner, Oscar Nierstrasz, and Jurgen J. Vinju. Engineering Academic Software (Dagstuhl Perspectives Workshop 16252). *Dagstuhl Reports*, 6(6):62–87, 2016.

[28] S. Guatelli, B. Mascialino, L. Moneta, I. Papadopoulos, A. Pfeiffer, M. G. Pia, and M. Piergentili. Experience with software process in physics projects. In *IEEE Symposium Conference Record Nuclear Science 2004.*, volume 4, pages 2100–2103 Vol. 4, Oct 2004.

[29] The Scrum Guide. Version 1. `https://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf`. Accessed: 01-15-19.

[30] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, May 2009.

[31] R. R. Harmon and N. Auseklis. Sustainable it services: Assessing the impact of green computing practices. In *PICMET '09 - 2009 Portland International Conference on Management of Engineering Technology*, pages 1707–1717, Aug 2009.

[32] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207 – 219, 2015.

[33] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207 – 219, 2015.

[34] M. A. Heroux, J. M. Willenbring, and M. N. Phenow. Improving the development process for cse software. In *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, pages 11–17, Feb 2007.

[35] Charles Hill. [title page]. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–1, Sep. 2016.

[36] L. Hochstein and V. R. Basili. The asc-alliance projects: A case study of large-scale parallel scientific code development. *Computer*, 41(3):50–58, March 2008.

[37] D. Hook and D. Kelly. Testing for trustworthiness in scientific software. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 59–64, May 2009.

[38] Upulee Kanewala and James M. Bieman. Techniques for testing scientific programs without an oracle. In *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, SE-CSE '13, pages 48–57. IEEE Press, 2013.

[39] Upulee Kanewala and James M. Bieman. Testing scientific software: A systematic literature review. *Information and Software Technology*, 56(10):1219 – 1232, 2014.

[40] Upulee Kanewala and James M. Bieman. Testing scientific software: A systematic literature review. *Inf. Softw. Technol.*, 56(10):1219–1232, October 2014.

[41] Daniel S. Katz, Sou-Cheng T. Choi, Hilmar Lapp, Ketan Maheshwari, Frank Löffler, Matthew Turk, Marcus Hanwell, Nancy Wilkins-Diehr, James Hetherington, James Howison, Shel Swenson, Gabrielle Allen, Anne Elster, Bruce Berriman, and Colin Venters. Summary of the first workshop on sustainable software for science: Practice and experiences (WSSSPE1). *Journal of Open Research Software*, 2(1), 2014.

[42] Daniel S. Katz, Sou-Cheng T. Choi, Kyle E. Niemeyer, James Hetherington, Frank Löffler, Dan Gunter, Ray Idaszak, Steven R. Brandt, Mark A. Miller, Sandra Gesing, Nick D. Jones, Nic Weber, Suresh Marru, Gabrielle Allen, Birgit Penzenstadler, Colin C. Venters, Ethan Davis, Lorraine Hwang, Ilian Todorov, Abani Patra, and Miguel de Val-Borro. Report on the third workshop on sustainable software for science: Practice and experiences (WSSSPE3). *Journal of Open Research Software*, 4(1):e37, 2016.

[43] Daniel S. Katz, Sou-Cheng T. Choi, Nancy Wilkins-Diehr, Neil Chue Hong, Colin C. Venters, James Howison, Frank J. Seinstra, Matthew Jones, Karen Cranston, Thomas L. Clune, Miguel de Val-Borro, and Richard Littauer. Report on the second workshop on sustainable software for science: Practice and experiences (WSSSPE2). *Journal of Open Research Software*, 4(1):e7, 2016.

[44] D. Kelly, D. Hook, and R. Sanders. Five recommended practices for computational scientists who write software. *Computing in Science Engineering*, 11(5):48–53, Sep. 2009.

[45] D. Kelly, S. Thorsteinson, and D. Hook. Scientific software testing: Analysis with four dimensions. *IEEE Software*, 28(3):84–90, May 2011.

[46] Diane Kelly, Rebecca S, Room Saint, Patrick Floor, Rebecca Sanders, and Diane Kelly. The challenge of testing scientific software. In *in Proc. Conf. for the Association for Software Testing*, pages 30–36, 2008.

[47] G. S. Lemos and E. Martins. Specification-guided golden run for analysis of robustness testing results. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 157–166, June 2012.

[48] Mika V. Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Softw. Eng.*, 35(3):430–448, May 2009.

[49] Simon McIntosh-Smith, Terry Wilson, Jon Crisp, Amaurys Ávila Ibarra, and Richard B. Sessions. Energy-aware metrics for benchmarking heterogeneous systems. *SIGMETRICS Perform. Eval. Rev.*, 38(4):88–94, March 2011.

[50] E. S. Mesh. Supporting scientific se process improvement. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 923–926, May 2015.

[51] Greg Miller. A scientist's nightmare: Software problem leads to five retractions. *Science*, 314(5807):1856–1857, 2006.

[52] R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180, March 2015.

[53] A. W. U. Munipala and S. V. Moore. Code complexity versus performance for gpu-accelerated scientific applications. In *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*, pages 50–50, Nov 2016.

[54] Christian Murphy, Mohammad Raunak, Andrew King, Sanjian Chen, Christopher Imbraino, Gail Kaiser, Insup Lee, Oleg Sokolsky, Lori Clarke, and Leon Osterweil. On effective testing of health care simulation software. *Technical Reports (CIS)*, 05 2011.

[55] Udit Nangia and Daniel S. Katz. Track 1 paper: Surveying the u.s. national postdoctoral association regarding software use and training in research, Aug 2017.

[56] Aziz Nanthaamornphong and Jeffrey C. Carver. Test-driven development in scientific software: a survey. *Software Quality Journal*, pages 1–30, 2015.

[57] Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 12:1–12:10. ACM, 2010.

[58] Danny Perez, Ekin Cubuk, Amos Waterland, Efthimios Kaxiras, and Arthur Voter. Long-time dynamics through parallel trajectory splicing. *Journal of Chemical Theory and Computation*, 12, 11 2015.

[59] D. E. Post and R. P. Kendall. Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from asci. *Int. J. High Perform. Comput. Appl.*, 18(4):399–416, November 2004.

[60] D. E. Post and R. P. Kendall. Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from asci. *The International Journal of High Performance Computing Applications*, 18(4):399–416, 2004.

[61] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016.

[62] Michael J Quinn. *Parallel computing: theory and practice*. McGraw-Hill, Inc., 1994.

[63] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, Nov 2012.

[64] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 541–550. ACM, 2008.

[65] Roger S. Pressman and Bruce R. Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2002.

[66] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 181–190. ACM, 2018.

[67] R. Sanders and D. Kelly. Dealing with risk in scientific software development. *IEEE Software*, 25(4):21–28, July 2008.

[68] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[69] Judith Segal. Some problems of professional end user developers. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 111–118, 2007.

[70] Judith Segal. Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community. *Computer Supported Cooperative Work (CSCW)*, 18(5):581, Sep 2009.

[71] Magnus Thorstein Sletholt, Jo Hannay, Dietmar Pfahl, Hans Christian Benestad, and Hans Petter Langtangen. A literature review of agile practices and their effects in scientific software development. In *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, SECSE '11, pages 1–9. ACM, 2011.

[72] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[73] A. Sutherland and G. Venolia. Can peer code reviews be exploited for later information needs? In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 259–262, May 2009.

[74] Sergiy A. Vilkomir, W. Thomas Swain, Jesse H. Poore, and Kevin T. Clarno. Modeling input space for testing scientific computational software: A case study. In Marian Bubak, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2008*, pages 291–300, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[75] Jan Vitek and Tomas Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 33–38. ACM, 2011.

[76] Peng Zhang, Jiuling Wang, Ali Farhadi, Martial Hebert, and Devi Parikh. Predicting failures of vision systems. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

APPENDIX A

SOFTWARE METRICS IN RESEARCH SOFTWARE

A.1 Specific Metrics Identified

This appendix provides more detail about the specific metrics that we grouped into the high-level categories in Table 2.2. The following list provides the specific metrics we grouped into each category. The numbers in parentheses represent how many respondents indicated knowledge of the metric and use of the metric, respectively. That is (1,0) indicates one person knew the metric, but no one actually used it. each metric was mentioned as known and as used, respectively.

- *Code Metrics*: afferent couplings (1,0), binary size (1,1), clarity (1,0), code evolution metrics (1,0), code to comment ratio (2,0), cohesiveness (3,0), comment density (1,1), coupling (7,3), cyclic dependency (1,0), cyclomatic complexity (16,3), defect density (3,1), depths (1,0), function points (4,0), Halstead programming effort (1,1), information entropy (1,0), lines of code (LOC) (40,8), McCabe (1,0), number of classes (1,0), number of clones (2,0), number of modules (2,0), and program size (2,0); and

- *General Quality Metrics*: accuracy (1,1), barrier of entry (1,1), code language (1,0), encapsulation (1, 0), feature usage count (1,0), formal correctness (1,1), interoperability (1,1), mailing list activity (1,0), maintainability (4,3), number of bugs (3,1), portability (3,3), reproducibility (1,1), sustainability (1,1), technical debt (1,0), and usability (3,3); and

- *Performance Metrics*: build time (1,1), compile time (2,0), computing (1,1), CPU (1,0), execution time (12,9), FLOPS (1,1), FLOPS per [US] dollar (1,1), memory footprint (2,1), memory usage (5,5), performance (10,9), resource usage monitoring (1,1), scalability (3,3), and scaling with problem size (1,1); and

- *Process Metrics*: amount of documentation (1,0), app launch count (1,1), authors/committers (1,0), cycle time (3,3), development hours/story (1,1), development man [person] years (1,0), documentation (1,0), feature delivered (1,0), files (1,0), forks (2,0), functionality (1,1), GitHub (1,0), JIRA to track development (1,0), number of commits (4,0), number of developers (1,0), productivity (1,1), recursive validation (1,1), Redmine project management (1,0), reliability (1,1), request count (2,0), size of ticket tracker (1,1), test failures (1,0), and volume of mailing list traffic (1,0); and

- *Recognition Metrics*: citations (4,3), downloads (5,3), number of users (4,1), number of projects adopting code [code adoption] (1,1), and page views (1,0); and

- *Testing Metrics*: bug tracking and monitoring (10,4), code coverage (17,12), days between failed tests (1,0), days to fix failing test (1,0), number of passing tests (2,2), number of platforms covered by tests (7,1), number of tests (1,1), test time (1,1), testability (3,0), and testing (2,2)

APPENDIX B

TESTING TECHNIQUES IN RESEARCH SOFTWARE

B.1 List of Familiar Testing Techniques

This appendix provides the list of testing techniques respondents mentioned they were familiar with in response to the survey question Q9. The numbers in the parenthesis represent how many respondents indicated that testing technique.

Unit testing (87), Integration testing (43), Regression testing (39), Continuous integration (33), Test-driven development (18), Code coverage (16), System testing (14), Fuzz testing (12), Acceptance testing (9), validation testing (9), Performance testing (6), mock testing (6), Memory testing (6), Static analysis (6), Functional testing (6), Mutation testing (5), black box testing (4), Assertions testing (3), Documentation checking (3), Dynamic testing (3), CTest (3), Formal Methods (2), End-to-end testing (2), Random input testing (2), White box testing (2), Method of manufacture solution (2), Property-based testing (2), Scientific testing (2), Manual testing (2), User testing (2), Smoke test (2), Approval testing (2), Visual studio (2), Code reviews (2), Usability testing (1), Install testing (1), Reliability testing (1), Compatibility Testing (1), Load testing (1), Agile (1), Stress test (1), doctests (1), test suites (1), Golden master testing (1), Physics testing (1), Benchmarking (1), Automatic test-case generation (1), Use case test (1), Concolic testing (1), equivalence class (1), n-version (1), Beta testing (1), Alpha testing (1), A/B testing (1), Security testing (1), Behavioral testing (1), Linting (1), Code style checking (1), Dependencies testing (1), Resolution testing (1), Comparison with analytical

solutions (1), Built environment testing (1), Contracts (1), Jenkins automated testing (1), License compliance (1), Exploratory tests (1), Correctness tests (1), run-time instrumentation and logging (1), Profiling (1), Deployment testing (1), Nightly (1), Release (1), DBC (1), reference runs on test datasets (1), Checklist testing (1), gtest (1), jUnit (1), Squish (1), Caliper (1), Bamboo (1), GitLab (1), Periodic testing (1), Pre-commit (1), Design by contract (1), Statistical testing (1), Builtd testing (1), Bit-for-bit (1), Engineering tests (1), Method of exact solutions (1), Answer testing (1), Behavior-Driven Developent (1), Coding standards (1), Accuracy testing (1), Monitoring production apps (1), Asan (1), Tsan (1), Msan (1), Penetration testing (1), Checksum (1).

APPENDIX C

INSTITUTIONAL REVIEW BOARD CERTIFICATION

Office for Research

Institutional Review Board for the
Protection of Human Subjects

December 2, 2015

Jeffrey C. Carver, Ph.D.
Associate Professor
Department of Computer Science
College of Engineering
The University of Alabama
Box 870290

Re: IRB # EX-15-CM-110 (Revision) "Scientific Software Metrics Survey"

Dear Dr. Carver:

The University of Alabama Institutional Review Board has reviewed the revision to your previously approved exempt protocol. The board has determined that the change does not affect the exempt status of your protocol.

Please remember that your approval period expires one year from the date of your original approval, September 23, 2015, not the date of this revision approval.

Should you need to submit any further correspondence regarding this proposal, please include the assigned IRB application number. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants.

Good luck with your research.

Sincerely,

Carpantato T. Myles, MSM, CIM, CIP
Director & Research Compliance Officer
Office for Research Compliance

358 Rose Administration Building
Box 870127
Tuscaloosa, Alabama 35487-0127
(205) 348-8461
FAX (205) 348-7189
TOLL FREE (877) 820-3066

THE UNIVERSITY OF
ALABAMA®
Office of the Vice President for
Research & Economic Development
Office for Research Compliance

November 6, 2017

Nasir Uddin Eisty
Department of Computer Science
College of Engineering
The University of Alabama
Box 870290

Re: IRB # EX-17-CM-076 "Understanding Code Review Processes in Scientific Software Development"

Dear Mr. Eisty:

The University of Alabama Institutional Review Board has granted approval for your proposed research. Your protocol has been given exempt approval according to 45 CFR part 46.101(b)(2) as outlined below:

(2) Research involving the use of educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures or observation of public behavior, unless:
(i) information obtained is recorded in such a manner that human subjects can be identified, directly or through identifiers linked to the subjects; and (ii) any disclosure of the human subjects' responses outside the research could reasonably place the subjects at risk of criminal or civil liability or be damaging to the subjects' financial standing, employability, or reputation.

Your application will expire on November 5, 2018. If your research will continue beyond this date, complete the relevant portions of Continuing Review and Closure From. If you wish to modify the application, complete the Modification of an Approved Protocol Form. When the study closes, complete the appropriate portions of FORM: Continuing Review and Closure.

Should you need to submit any further correspondence regarding this proposal, please include the assigned IRB application number.

Good luck with your research.

Sincerely,

Carpantato T. Myles, MSM, CIM, CIP
Director & Research Compliance Officer
Office for Research Compliance

358 Rose Administration Building | Box 870127 | Tuscaloosa, AL 35487-0127
205-348-8461 | Fax 205-348-7189 | Toll Free 1-877-820-3066

THE UNIVERSITY OF ALABAMA® | Office of the Vice President for Research & Economic Development | Office for Research Compliance

August 20, 2019

Nasir Eisty
Computer Science
Box 870290

Re: IRB # EX-19-CM-171: "Understanding the Testing Process in Research Software Engineering"

Dear Mr. Eisty,

The University of Alabama Institutional Review Board has granted approval for your proposed research. Your application has been given exempt approval according to 45 CFR part 46. Approval has been given under exempt review category 2 as outlined below:

(2) Research that only includes interactions involving educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures, or observation of public behavior (including visual or auditory recording) if: (i) The information obtained is recorded by the investigator in such a manner that the identity of the human subjects cannot readily be ascertained, directly or through identifiers linked to the subjects.

The approval for your application will lapse on August 19, 2020. If your research will continue beyond this date, please submit the annual report to the IRB as required by University policy before the lapse. Please note, any modifications made in research design, methodology, or procedures must be submitted to and approved by the IRB before implementation. Please submit a final report form when the study is complete.

Please use reproductions of the IRB approved informed consent form to obtain consent from your participants.

Sincerely,

Carpantato T. Myles, MSM, CIM, CIP
Director & Research Compliance Officer

cc: Dr. Jeffrey Carver

358 Rose Administration Building | Box 870127 | Tuscaloosa, AL 35487-0127
205-348-8461 | Fax 205-348-7189 | Toll Free 1-877-820-3066