10-29-2018

A Survey of Software Metric Use in Research Software Development

Nasir U. Eisty
University of Alabama - Tuscaloosa

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Jeffrey C. Carver
University of Alabama - Tuscaloosa

Recommended Citation

N. U. Eisty, G. K. Thiruvathukal and J. C. Carver, "A Survey of Software Metric Use in Research Software Development," 2018 IEEE 14th International Conference on e-Science (e-Science), Amsterdam, Netherlands, 2018, pp. 212-222. doi: 10.1109/eScience.2018.00036

A Survey of Software Metric Use in Research Software Development

Nasir U. Eisty
Department of Computer Science
University of Alabama
Tuscaloosa, AL, USA
Email: neisty@crimson.ua.edu

George K. Thiruvathukal
Department of Computer Science
Loyola University Chicago
Chicago, IL, USA
Email: gkt@cs.luc.edu

Jeffrey C. Carver
Department of Computer Science
University of Alabama
Tuscaloosa, AL, USA
Email: carver@cs.ua.edu

***Abstract*—*Background*: Breakthroughs in research increasingly depend on complex software libraries, tools, and applications aimed at supporting specific science, engineering, business, or humanities disciplines. The complexity and criticality of this software motivate the need for ensuring quality and reliability. Software metrics are a key tool for assessing, measuring, and understanding software quality and reliability. *Aims*: The goal of this work is to better understand how research software developers use traditional software engineering concepts, like metrics, to support and evaluate both the software and the software development process. One key aspect of this goal is to identify how the set of metrics relevant to research software corresponds to the metrics commonly used in traditional software engineering. *Method*: We surveyed research software developers to gather information about their knowledge and use of code metrics and software process metrics. We also analyzed the influence of demographics (project size, development role, and development stage) on these metrics. *Results*: The survey results, from 129 respondents, indicate that respondents have a general knowledge of metrics. However, their knowledge of specific SE metrics is lacking, their use even more limited. The most used metrics relate to performance and testing. Even though code complexity often poses a significant challenge to research software development, respondents did not indicate much use of code metrics. *Conclusions*: Research software developers appear to be interested and see some value in software metrics but may be encountering roadblocks when trying to use them. Further study is needed to determine the extent to which these metrics could provide value in continuous process improvement.**

***Index Terms*—Survey, Software Metrics, Software Engineering, Research Software**

I. Introduction

Researchers in a number of scientific, engineering, business, and humanities domains increasingly develop and/or use software to conduct or support their research. Collectively, we refer to the software (libraries, tools, and applications) developed by these researchers as *research software*. These researchers were first described in the literature as *research software engineers (RSEs) [1]*. RSEs play a major role in defininining and designing the research software and seek recognition for their efforts.

Researchers draw insights and make critical decisions, at least partially, based upon results obtained from research software. The correctness of the design and implementation of this software is therefore of utmost importance. Low quality software is likely to produce less trustworthy results and may lead to incorrect research conclusions or engineering/design decisions.

We have observed that research software engineers often place less importance on traditional views of software quality and maintainability than on other scientific goals. Examples from the literature (Section II) confirm that there have been many efforts to understand how software engineering (SE) 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, most of these efforts have not led to research software teams that value understanding and measuring software quality over time.

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 paper. We refer readers to authoritative texts on the subject [4].

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).

Therefore in this paper we look beyond our anecdotal experiences in order to attain a deeper understanding of perceptions about software metrics directly from research software developers. The primary objective of this study is *to understand research software developers' knowledge and use of software metrics*. Ultimately, we desire to understand which metrics should be included in a software metrics suite specifically designed to support research software development.

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 type of software metrics have had on their respective projects.

The primary contributions of this paper 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.

The remainder of this paper is organized as follows. In Section II we describe previous work to motivate a series of research questions explored in this study. In Section III we explain the survey design. In Section IV we provide the detailed survey results. In Section V we discuss and interpret the survey results. In Section VI we enumerate the validity threats. In Section VII we draw conclusions.

II. 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.

A. 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* [6]. This interest increases the relevance of concepts like energy costs and sustainability. Specifically, some research software developers focus on *energy efficiency* and *carbon emissions* [11].

Third, in some subdisciplines of research software (e.g. simulation and modeling), 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 [17].

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 at very large (i.e. 10,000 node) computers [15]. 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 [18].

Finally, research software engineers (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 engineers, who not only produce code but produce tools that help others to do research [1],.

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 [2]. 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 [7] including:

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

[1] http://top500.org

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*** *– Which metrics are most commonly used?*
- ***RQ3*** *– What is the relationship between knowledge of metrics and their perceived usefulness?*

B. 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 [14], 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 [12]. 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 [16]. 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:

- ***RQ4*** *– Do research software developers perceive code complexity as a problem?*
- ***RQ5*** *– Is the frequency of complexity problems related to the use of or perceived helpfulness of metrics?*

III. Survey Design

To answer each research question defined in Section II, 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 1 shows the portion of the survey that we include in this analysis. The figure contains 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* [3] 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.

IV. Results

In this section, we present the results of the survey organized around the survey themes. In the subsequent discussion section, we use these results to answer the research questions. In total, we received 129 responses to the survey. Note that throughout this discussion, the survey questions refer to the numbers in Figure 1.

A. 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 (i.e. we exclude partial responses).

Fig. 1. Survey Questions

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]

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

2) Project Size: In Figure 2 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 [2] (see Section II-A), 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.

3) 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 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 3 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.

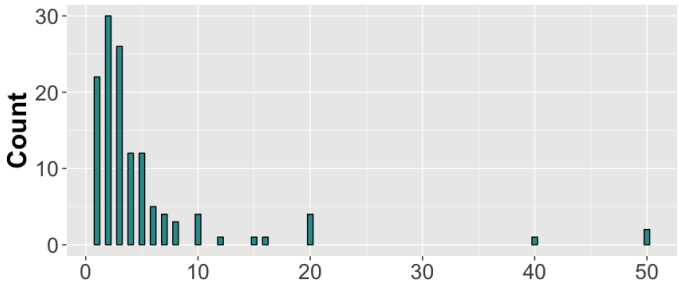

Fig. 2. Number of Developers

4) 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

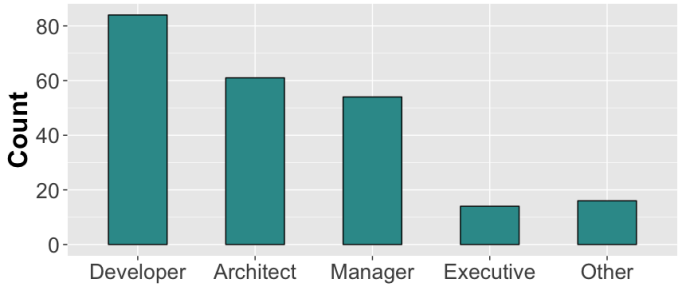

Fig. 3. Respondents' Role on Project

Figure 4, 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.

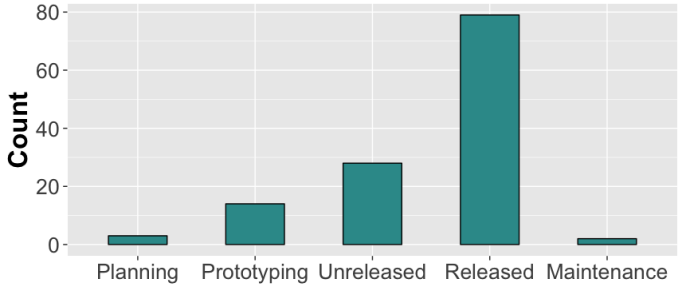

Fig. 4. Project Stage

B. Overall Analysis

Regarding the respondents' general knowledge of metrics (Question MQ1), the majority indicated they had *low* or *very low* knowledge of metrics (Figure 5). 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 6).

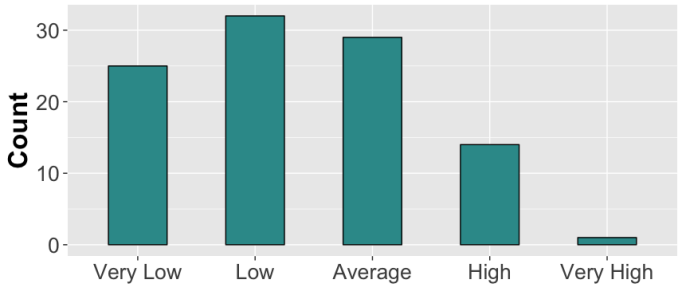

Fig. 5. Knowledge of Metrics

One would expect a relationship between general knowledge of metrics and perceived usefulness of those metrics. To

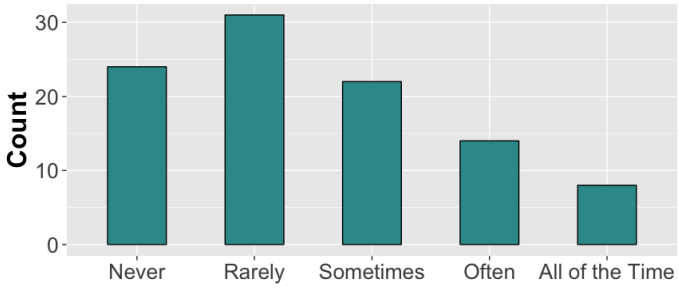

Fig. 6. Perceived Usefulness of Metrics

TABLE I
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

determine whether this relationship is present in our results, Table I shows the comparison of the respondents' general knowledge of metrics (Figure 5) with their perception of the usefulness of those metrics (Figure 6). 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.

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, porta-*

bility, 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 [5], [8], [9], [10]. Table II provides an overview of the responses.

TABLE II
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

Finally, in Figure 7, 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.

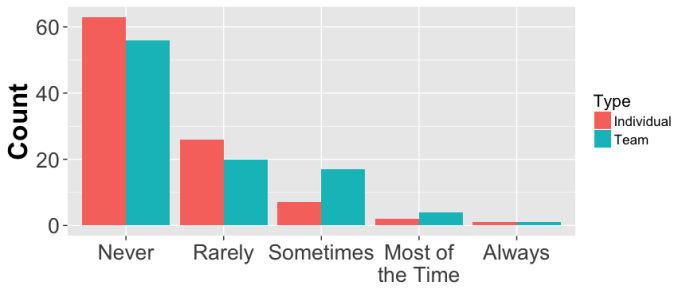

Fig. 7. Frequency of using Metrics for Evaluation

Similar to the analysis above, we analyzed whether there was a relationship between perceived usefulness of metrics (Figure 6) and the use of metrics for evaluation (Figure 7). 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.

C. Influence of Demographics

In this section, we examine whether the demographics defined in Section IV-A 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).

1) 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 8) and see metrics as less useful (Figure 9) 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 suprising given the results in Figures 5 and 6.

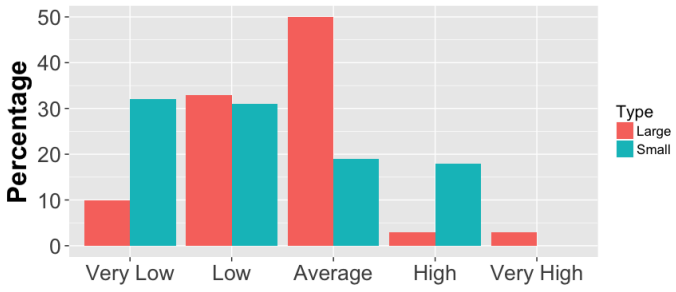

Fig. 8. Influence of project size on overall knowledge of metrics

Turning to specific metrics, larger teams have knowledge of more metrics in four of the six categories (Figure 10). Conversely, use of metrics is more consistent between large and small teams with two notable exceptions. Large teams use

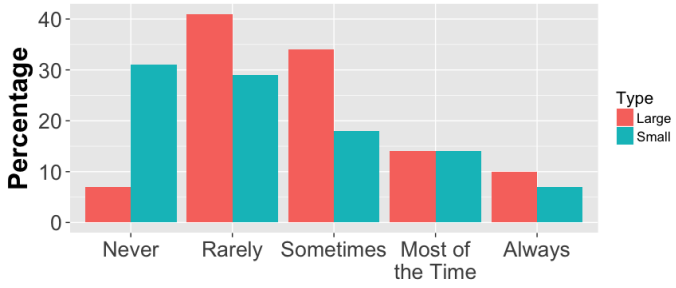

Fig. 9. Influence of project size on percieved usefulness of metrics

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.

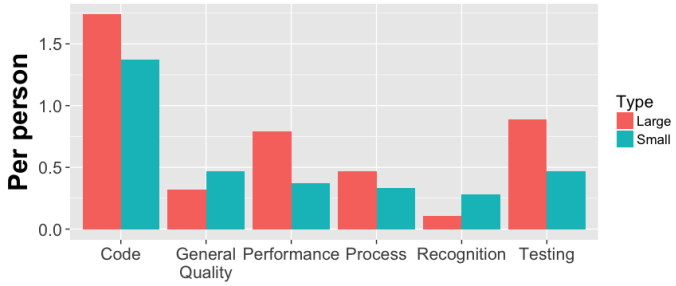

Fig. 10. Influence of project size on known metrics

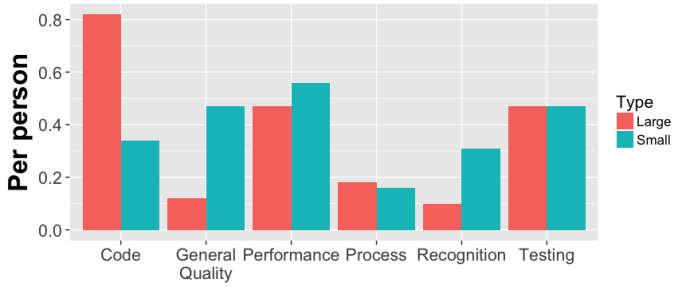

Fig. 11. Influence of project size on used metrics

2) 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 IV-A3), 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.

3) 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 12) and overall perception of usefulness (Figure 13) 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.

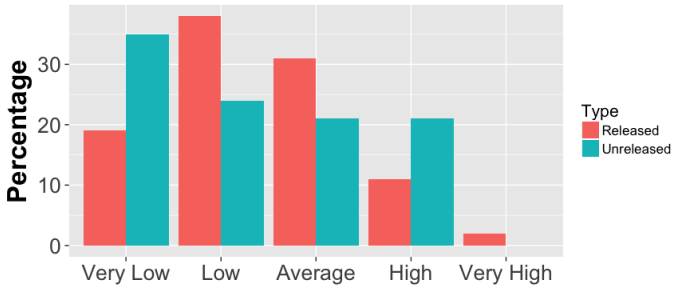

Fig. 12. Influence of project stage on overall knowledge

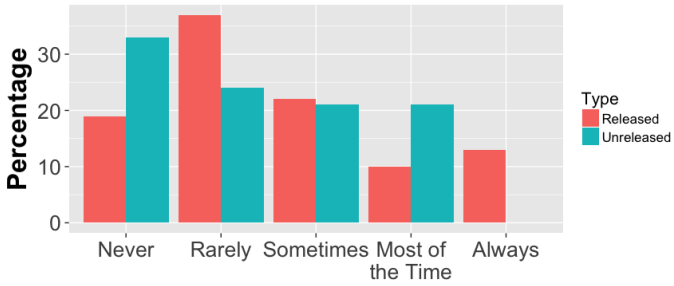

Fig. 13. Influence of project stage on perceived usefulness

For knowledge of specific types of metrics, project stage had very little differentiating effect. Conversely, for use of specific types of metrics (Figure 14), respondents from unreleased software found *performance* metrics more useful while respondents from released software found *testing* metrics more useful.

D. Code Complexity

The survey respondents perceived code complexity to be a problem. In Figure 15, 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 16), the vast majority said they *never* used code complexity metrics, with only a small number using them more often than *rarely*.

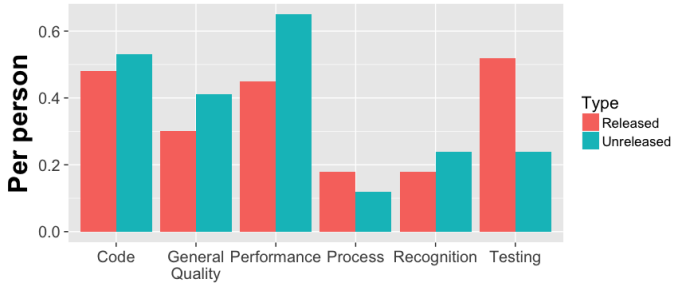

Fig. 14. Influence of project stage on specific metrics used

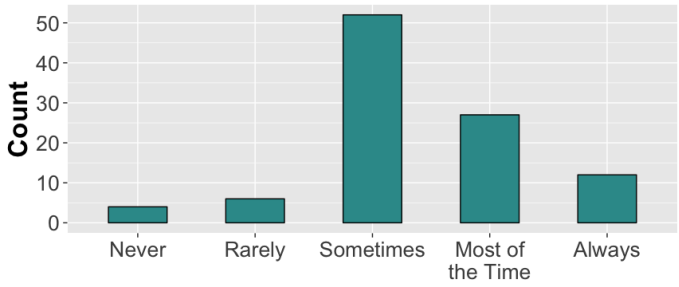

Fig. 15. Frequency of Code Complexity Problems

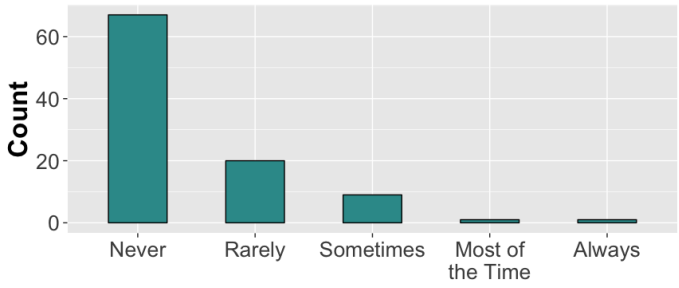

Fig. 16. Use of Code Complexity Metrics

Of those that used complexity metrics (e.g. rarely or above in Figure 16), the responses to CQ3 indicated that the majority found the metrics were *never* useful (Figure 17). Finally, there was no significant relationship between the frequency of complexity problems and the use or helpfulness of complexity metrics.

V. Discussion

In this section we discuss key insights related to each research question.

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

In general, the majority of respondents reported *very low* to *low* knowledge of metrics, while just under half indicated that metrics are *rarely* or *never* useful. However, respondents

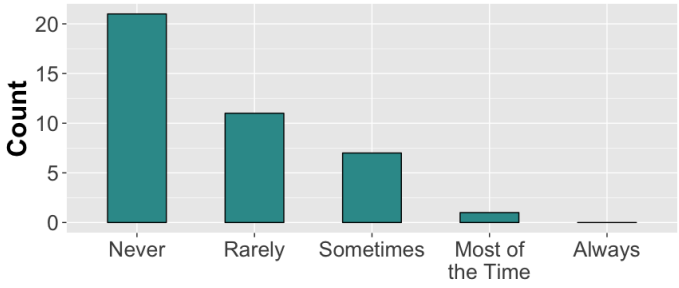

Fig. 17. Usefulness of Code Complexity Metrics

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 [4].

RQ2 – *Which metrics are most commonly used?*

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 I show that respondents reported the most unique code metrics and reported the highest frequency of 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 [4] and TDD [13].
- *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 [8], [9], [10].

- *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.

RQ3 *– What is the relationship between knowledge of metrics and their perceived usefulness?*

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.

RQ4 *– Do research developers perceive code complexity as a problem?*

Most survey respondents indicated that code complexity is at least *sometimes* a problem. While this result was not entirely a surprise, given our experiences of observing 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.

RQ5 *– Is the frequency of complexity problems related to the use of or perceived helpfulness of metrics?*

Although research software developers report code complexity as problematic, they *rarely* or *never* consider complexity metrics to be useful. This result is surprising (and somewhat troubling), given the prevalence of code complexity problems. Additionally, the results showed no significant relationship between the frequency of complexity problems and helpfulness of complexity metrics. 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.

VI. Threats

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

A. 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.

B. 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.

C. 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.

VII. Conclusion

In this paper, we report on the results of a survey of research software developers to assess knowledge and use of software metrics. In all 129 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.

The frequent knowledge and use of testing metrics is a bit more surprising. One potential explanation for this observation is that, according to the literature, many research software projects unconsciously embrace agile processes, which emphasis test-driven development (TDD) as part of their process. We need to conduct further study to understand more about the software development process used by research software development teams to identify which particular metrics would be the most useful. Although the survey responses suggest a significant number of teams that use agile processes, further study would be required to understand the prevalence of agile vs. other processes and how this influences metrics perceptions and use.

In conclusion, this work shows that various software metrics 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 that are based upon the SE methods that they already employ in their projects (development process and testing).

Acknowledgments

We thank the survey respondents. Carver and Eisty acknowledge support from NSF-1445344. Thiruvathukal acknowledges support from NSF-1445347.

Appendix

Specific Metrics Identified

This appendix provides more detail about the specific metrics that we grouped into the high-level categories in Table II. 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)

REFERENCES

[1] R. Baxter, N. Chue Hong, D. Gorissen, J. Hetherington, and I. Todorov. The research software engineer. 9 2012. Digital Research 2012 ; Conference date: 10-09-2012 Through 12-09-2012.

[2] 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.

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

[4] N. Fenton and J. 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, Oct. 2014.

[5] C. Goble, J. Howison, C. Kirchner, O. Nierstrasz, and J. J. Vinju. Engineering Academic Software (Dagstuhl Perspectives Workshop 16252). *Dagstuhl Reports*, 6(6):62–87, 2016.

[6] 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.

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

[8] D. S. Katz, S.-C. T. Choi, H. Lapp, K. Maheshwari, F. Löffler, M. Turk, M. Hanwell, N. Wilkins-Diehr, J. Hetherington, J. Howison, S. Swenson, G. Allen, A. Elster, B. Berriman, and C. Venters. Summary of the first workshop on sustainable software for science: Practice and experiences (WSSSPE1). *Journal of Open Research Software*, 2(1), 2014.

[9] D. S. Katz, S. T. Choi, K. E. Niemeyer, J. Hetherington, F. Löffler, D. Gunter, R. Idaszak, S. R. Brandt, M. A. Miller, S. Gesing, N. D. Jones, N. Weber, S. Marru, G. Allen, B. Penzenstadler, C. C. Venters, E. Davis, L. Hwang, I. Todorov, A. Patra, and M. 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.

[10] D. S. Katz, S. T. Choi, N. Wilkins-Diehr, N. Chue Hong, C. C. Venters, J. Howison, F. J. Seinstra, M. Jones, K. Cranston, T. L. Clune, M. de Val-Borro, and R. Littauer. Report on the second workshop on sustainable software for science: Practice and experiences (WSSSPE2). *Journal of Open Research Software*, 4(1):e7, 2016.

[11] S. McIntosh-Smith, T. Wilson, J. Crisp, A. A. Ibarra, and R. B. Sessions. Energy-aware metrics for benchmarking heterogeneous systems. *SIGMETRICS Perform. Eval. Rev.*, 38(4):88–94, Mar. 2011.

[12] 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.

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

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

[15] 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.

[16] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[17] J. Vitek and T. Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 33–38, New York, NY, USA, 2011. ACM.

[18] P. Zhang, J. Wang, A. Farhadi, M. Hebert, and D. Parikh. Predicting failures of vision systems. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.