Evaluating the Sustainability of Computational Science and Engineering Software: Empirical Observations

James M. Willenbring
Software Engineering & Research Department
Sandia National Laboratories*
Albuquerque, New Mexico
jmwille@sandia.gov

Gursimran Singh Walia
School of Computer and Cyber Sciences
Augusta University
Augusta, Georgia
gwalia@augusta.edu

***Abstract*— Software sustainability is critical for Computational Science and Engineering (CSE) software. It is also challenging due to factors ranging from funding models to the typical lifecycle of a research code to the inherent challenges of running fast on the newest architectures. Furthermore, measuring sustainability is challenging because sustainability consists of many complex attributes. To identify useful metrics for measuring CSE software sustainability, we gathered data from multiple freely available sources, including GitHub, SLOCCount, and Metrix++. This paper discusses the challenges practitioners face when measuring the sustainability of CSE software. We present an analysis of data with associated observations and future directions to better understand CSE software sustainability and how this work can be used to support decisions and improve sustainability by observing trends in metrics over time.**

I. INTRODUCTION

Software sustainability is a key issue in the Computational Science and Engineering (CSE) domain. CSE software projects often begin as a research activity. Software engineering concerns are secondary to the research objectives driving the project [4]. While some research activities fail (as is consistent with research), successful projects often result in software with a very long useful life. So while there is a risk that investing early on in sustainability may prove to be a wasted effort in a sense if the research activity fails, the penalty for not investing in sustainability for initially successful projects is high. Projects developed without sustainability in mind eventually become fragile. For example, poor designs limit extensibility and evolvability, and insufficient testing leads to a lack of maintainability. Other factors (e.g., funding models, developer training, staffing challenges, etc.) can also contribute to a lack of sustainability [2].

Research and practice indicate that sustaining CSE software is inherently complex, as discussed herewith. CSE software utilizes cutting-edge algorithms and language features to promote performance on the world's largest supercomputers, particularly on new architectures. Codes are often significant - hundreds of thousands or millions of code lines, and commonly use several third-party software packages (many of which are open source). While these challenges are not new, they have grown in recent years. One of the author's previous work offered some simple ways to improve the quality of CSE software [4]. Our suggestions included the use of source code management, issue tracking, documented processes, source-centric documentation, pair programming, continuous process improvement, and other software practices. While the principles remain relevant, developers' environment has evolved to be much more complex.

Prior work lists several sustainability attributes: extensibility, interoperability, maintainability, portability, reusability, scalability, and usability [10]. Motivated by an array of literature on sustainability attributes, this work analyzes the most critical aspects of software sustainability in the context of CSE software. Several groups have gathered, analyzed, and/or defined metrics with a focus towards software sustainability or an attribute of software sustainability [15] [7] [5] [9] [8] [1]. However, previous efforts have not closely examined metrics in the context of CSE software sustainability. Furthermore, prior literature lacks tool support for researchers in gathering more advanced metrics, such as those in [7] for highly complicated (C++) codebases.

This research aims to charazterize better and identify barriers to CSE software sustainability, as well as reduce those barriers by identifying tools and techniques to support decision making focused on improving software quality.

II. APPROACH

We analyzed CSE software projects openly available on GitHub, GitLab, and Bitbucket. Although not all of the repositories go back to the beginning of the projects (some first used another version control system and snapshotted the code into git), all of the repositories have substantial history to examine. Additionally, due to scientific software community involvement, we have access to the contributors to many of the projects being analyzed.

Our strategy for measuring sustainability focuses on the various component attributes before embarking on a more

**Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the US Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.*
This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. SAND2021-7711 C.

DOI reference number: 10.18293/SEKE2022-154.

holistic analysis. An initial broad set of representative CSE software projects was carefully selected for the study. We started by considering software that comprises the Extreme-scale Scientific Software Development Kit (xSDK). The xSDK [14] project was created to improve the interoperability and sustainability of scientific libraries that are common dependencies for scientific software.

Next, we chose packages that represented both large and small source code bases and development teams and a variety of primary development institutions (four US national laboratories and two universities). We also wanted projects with lengthy histories, which would allow us to go back and look at changes over time. Six out of the seven projects chosen are also currently funded in part under Math Libraries within the Exascale Computing Project's (ECP) Software Technologies (ST) thrust [3]. A group of experts initially chose this software group to be part of ECP based on each software package's current and potential value to high-performance computing. Further, ECP ST is very interested in improving the sustainability of this software. Below is a brief description of the seven software projects chosen for our metric collection activity. The names of the projects have been changed to guard against unintended conclusions being drawn about the sustainability of any specific project.

Project 1 includes linear and non-linear solvers as well as preconditioners for partial differential equation-based systems of equations. **Project 2** is a collection of solvers and enabling technologies used for large-scale, complex multi-physics engineering and scientific problems. **Project 3** is a distributed memory direct LU solver for non-symmetric sparse linear systems of equations. **Project 4** provides algebraic multigrid sparse preconditioners and solvers. **Project 5** is a dense linear algebra library that provides linear, least squares, eigen, and S.V.D. solvers. **Project 6** is a finite element and adaptive mesh refinement code. **Project 7** is a sparse linear and nonlinear eigenvalue solver package.

TABLE 1: Language, SLOC, Contributors & Commits

Package	Language	SLOC	Contribs	Commits
Project 1	C 83%	796123	198	82663
Project 2	C++ 82%	4179781	250	95384
Project 3	C 96%	80752	14	645
Project 4	C 81%	441057	37	11587
Project 5	C++ 45%	317082	51	8086
Project 6	C++ 100%	293062	114	14668
Project 7	C 91%	109559	26	9045

The purpose of gathering these metrics is to analyze the correlation of the metrics with aspects of sustainability. Because no single metric fully reflects the sustainability of a software project, we look at several metrics, including some directly related to the source code, such as complexity and lines of code, and others that are not directly related to the code itself, such as a number of commits and contributors. Additionally, we also analyzed metric trends over time. Researchers and practitioners can utilize trends to better understand if a codebase is becoming more or less sustainable (e.g. observing an increasing or decreasing number of contributors, cyclomatic complexity, or maintenance index).

III. DATA ANALYSIS AND RESULTS

Our approach involved gathering metrics from accessible sources and tools. The first set of results consists of metrics gathered from development snapshots of the seven codes we chose from May 2021. The second set of results includes metrics collected from five snapshots in time for each of the seven codes from May 2017-2021 (once per year). We describe each of these sets of metrics below.

The first set of results obtained information about the number of contributors and the number of commits from GitHub and Gitlab, and from the git command line (as Bitbucket does not supply this information). We also gathered metrics involving the use of the tool SLOCCount [11] to obtain the primary programming language for each project, along with the percentage of lines in the project of the primary language and the total number of source lines of code (SLOC).

The second set of metrics was collected using Metrix++ version 1.7.0 [6]. To capture yearly snapshots, git commands of the form *git checkout `git rev-list -n 1 --first-parent --before="2019-05-24 00:00" <primary_branch_name>* were used. The metrics included in this set are:

- *Maximum Complexity*: The maximum cyclomatic complexity found in the code.
- *Average Complexity*: The average cyclomatic complexity found in all regions of the code.
- *Lines of code*: The total number of lines of code. Note Metrix++ considers only C, C++, and Java code (not Fortran, Python, or scripts).
- *Maintenance Index:* A measure of maintainability computed from cyclomatic complexity and lines of code. A lower value indicates a higher level of maintainability.

TABLE 2: Max Cyclomatic Compexity

Package	2017	2018	2019	2020	2021
Project 1	1786	1788	2319	540	540
Project 2	648	648	648	547	547
Project 3	202	204	301	301	301
Project 4	649	765	815	877	913
Project 5	261	261	261	261	261
Project 6	114	114	137	134	238
Project 7	86	86	86	86	83

TABLE 3: Average Cyclomatic Compexity

Package	2017	2018	2019	2020	2021
Project 1	4.63	4.69	4.88	4.69	4.58
Project 2	2.59	2.47	2.46	2.39	2.30
Project 3	12.50	11.97	10.62	10.31	10.36
Project 4	7.14	6.65	6.50	6.48	6.33
Project 5	5.45	5.40	5.27	5.30	5.33
Project 6	2.34	2.33	2.14	2.48	2.49
Project 7	3.95	3.97	4.14	4.11	4.10

TABLE 4: Lines of Code (in 1,000's)

Package	2017	2018	2019	2020	2021
Project 1	483	525	584	613	657
Project 2	2682	3082	3076	3279	3414
Project 3	55	58	79	81	86
Project 4	410	359	365	390	405
Project 5	128	131	136	137	138
Project 6	107	122	154	216	284
Project 7	71	79	84	89	95

TABLE 5: Maintenance index computed by Metrix++ using complexity and lines of code data

Package	2017	2018	2019	2020	2021
Project 1	1.40	1.41	1.43	1.42	1.41
Project 2	1.21	1.18	1.18	1.19	1.18
Project 3	2.35	2.29	2.08	2.05	2.08
Project 4	1.81	1.78	1.77	1.79	1.79
Project 5	1.76	1.71	1.70	1.70	1.71
Project 6	1.21	1.21	1.19	1.24	1.24
Project 7	1.31	1.32	1.33	1.33	1.33

IV. DISCUSSION OF RESULTS

We discuss significant results concerning metrics reported in the previous section focused around key themes and contributions to understanding CSE software sustainability.

SLOC and Contributors: - Poor software design, for example, poor understandability, has a greater than linear impact as SLOC increases in terms of maintainability and evolvability. Code size metrics such as SLOC can be useful to measure over time. For example, SLOC growth in excess of feature set growth may indicate the need to refactor.

A very low number of contributors can be a sustainability risk in that the knowledge of the code is owned by a small group of people. The number of contributors to the seven codes varies by more than an order of magnitude. Further, the ratio of SLOC to contributors may speak to maintainability. Less code per contributor means fewer lines that each contributor needs to maintain. Projects 2 and 4 have significantly higher SLOC to contributor ratios than the other five codes.

Complexity: A lower average complexity should enhance readability and maintainability, all else equal. Interestingly, the average complexity of Project 3 is nearly twice the average complexity of the next highest sample code.

TABLE 6: SLOC and Contributors

Package	SLOC	contributors	SLOC/contrib
Project 1	796123	198	4021
Project 2	4179781	250	16719
Project 3	80752	14	5768
Project 4	441057	37	11920
Project 5	317082	51	6217
Project 6	293062	114	2571
Project 7	109559	26	4214

While cyclomatic complexity does not capture all aspects of maintainability, by definition, it does reflect the number of paths through the code. If this value is growing over time, it can increase the maintenance burden. The Max complexity data gathered in Table 2 is interesting in that for three codes the value grew between 2017 and 2021, for two the value fell, and for two it remained nearly or exactly the same. Metrix++ has a "*hotspot*" feature that allows a person to identify regions with a complexity greater than a given *threshold*, which can be used to support a targeted refactoring effort.

Maintenance Index: The Maintenance index data in Table 5 does not change dramatically for any packages over time. This is again not surprising because these are large, established code bases, and in any given year, large portions of the codebase do not change. The most significant change in the codes' value comes from Project 3 between 2018 and 2019, with the Maintenance index falling from 2.29 to 2.08. We note that in the same period, Table 4 shows that the Lines of code increased substantially from 58,034 to 79,069. It is reasonable that a 36% increase in the size of the codebase would cause a noticeable decrease in the maintainability index if the new code was written more maintainably.

Similar to the previous observations for complexity, we feel that looking at changes in the maintenance index both over time and addressing maintenance "hotspots" could improve the maintainability and sustainability of codes. In addition, these checks can be automated in continuous integration or nightly processes and tracked over time to identify trends.

Metrix++ Hotspot Feature: As mentioned above, the hotspot feature in Metrix++ is a useful tool that allows regions of code to be identified that exhibit a metric value above a user-specified threshold. The tool is both simple to use and powerful. For example, the below command identified five regions of code in Project 4 with cyclomatic complexity equal to or greater than 500: *metrix++ limit --db-file=proj4.2019.lines.complex.maint.db --max-limit=std.code.complexity:cyclomatic:500*

The tool supports more advanced features that allow it to be used in an automated testing environment. The return code of the limit function is equal to the number of instances in the code where the metric threshold specified is exceeded. Therefore, automated, pre-push testing can prevent complex code from being added by checking this return code. Alternatively, automated metrics can be gathered and provided to code reviewers to use in their analysis.

While finding all areas of high complexity might be useful in some contexts, often it is preferable to consider only new or modified (the term Metrix++ uses for this option is "touched") code. For those cases, specify a previous version of a database file (using *--db-file-prev*) to compare against: *metrix++ limit --db-file=proj4.2019.lines.complex.maint.db --db-file-prev=../2018-05-24/proj4.2018.lines.complex.maint.db --max-limit=std.code.complexity:cyclomatic:500 --warn-mode=touched*

In our example, three of the five regions were touched. Finally, because refactoring all code that is touch may not be practical, Metrix++ supports a "*trend*" feature (using *--warn-mode=trend*, rather than *touched*) that only identifies code

for which the metric in question has gotten worse since the previous state. In our example, two of the three touched regions exhibited a negative trend.

In summary, the hotspot feature in Metrix++ allows a team to not only identify regions of their code of concern (for example, due to high complexity or maintenance index), but also allows the team to automate the tracking and even prevention of additional regions of concern. This feature can be used to support maintainability and sustainability.

V. CONCLUSION AND RELEVANCE TO INDUSTRY

While it is never appropriate to make broad conclusions based on a single metric, the metrics we studied provide quantitative data that can be used to support decisions. We caution, for example, against using any simple metrics to claim that one code is more sustainable than another. That said, a developer performing a code review can benefit from using the Metrix++ hotspot feature by considering changes in maintenance index or max or average complexity metrics in the broader context of the proposed changes.

By sampling multiple metrics for a single code base over time, practitioners can glean whether it is becoming more or less sustainable. Such information may help assess the effectiveness of changing development practices or tools. For example, a decreasing maintenance index for a code base following the adoption of a new development practice supports the hypothesis that the new practice is beneficial.

A similar approach could be used in evaluating the impact of a refactoring effort. For example, before refactoring a large chunk of code, one might gather average and maximum complexity as well as lines of code and maintenance index metrics. Then, the areas of complexity higher than a given threshold could be identified as candidates for refactoring. After the refactoring step, the metrics can be taken again to help quantify the impact of the refactoring, and metrics could be sampled periodically to understand the effects of development practices on the sustainability of the codebase.

In the future, we plan to explore sustainability factors not addressed by source code metrics, such as sustainability of dependencies, and how the sustainability of CSE software is impacted by the sustainability of the CSE software ecosystem as a whole [12]. For example, consider how common interfaces could improve software sustainability.

Another area of future work would be to consider smaller logical subsets of codebases and "hotspots" identified using Metrix++. We could also study other metrics such as contributors at a more granular level. Specifically, we could compare the number of frequent and recent contributors and total contributors to functionality in the code that is effectively orphaned (no currently assigned developers) to functionality that is more actively supported.

In an effort to help code teams and project leadership gather and effectively utilize metrics and related tools in their scientific software development efforts, we are also forming a Software Development Kit (SDK) community in the area of Tools for Code Mining and Data Analysis [13].

This research effort can make significant contributions to the understanding of the sustainability of relevant components of the CSE software stack. We analyzed the seven code projects part of the xSDK, six of which are part of the US DOE Exascale Computing Project. This allows us to base our findings on industrial representative CSE software, increasing the generalizability of our results.

Perhaps most importantly, a better understanding of software sustainability can help to identify how to design from the onset for sustainability, which has the potential to save significant developer time (and money) and prevent a lot of frustration dealing with unsustainable code.

Bibliography

[1] Bouwers, E., van Deursen, A., & Visser, J. (2013). Evaluating Usefulness of Software Metrics: An Industrial Experience Report. *Proc. Int'l Conf. Software Eng. (ICSE 13), IEEE*, 921-930.

[2] Heroux, M. A., & Allen, G. (2016, Sept). Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges Workshop Report. *Networking and Information Technology Research and Development (NITRD) Program*.

[3] Heroux, M. A., Carter, J., Thakur, R., McInnes, L., Ahrens, J., Munson, T., & Neeley, J. R. (2020, February 1). ECP Software Technology Capability Assessment Report. 10.2172/1606665

[4] Heroux, M. A., & Willenbring, J. M. (2009). Barely sufficient software engineering: 10 practices to improve your CSE software. *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 15-21. 10.1109/SECSE.2009.5069157

[5] Koziolek, H. (2011). Sustainability evaluation of software architectures: A systematic review. *Proceedings of the Joint ACM SIGSOFT Conference - QoSA and ACM SIGSOFT Symposium - ISARCS on Quality of Software Architectures - QoSA and Architecting Critical Systems - ISARCS QoSA-ISARCS '11*, 3-12.

[6] *Metrix++ Web Page*. (n.d.). https://metrixplusplus.github.io/metrixplusplus/

[7] Sarkar, S., Kak, A., & Rama, G. (2008). Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software. *IEEE Transactions on Software Engineering*, *34*(5), 700-720.

[8] Sarkar, S., Rama, G. M., & Kak, A. C. (2007). API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. *IEEE Trans. Software Eng.*, *33*(1), 14-32.

[9] Sehestedt, S., Cheng, C.-H., & Bouwers, E. (2014). Towards quantitative metrics for architecture models. *In Proceedings of the WICSA 2014 Companion Volume (WICSA '14 Companion). ACM*, Article 5, 4 pages. http://dx.doi.org/10.1145/2578128.2578226

[10] Venters, C. C., Lau, L., Griffiths, M. K., Holmes, V., Ward, R. R., Jay, C., & J, X. (2014). The Blind Men and the Elephant: Towards an Empirical Evaluation Framework for Software Sustainability. *Journal of Open Research Software*, *2(1)*(8). http://doi.org/10.5334/jors.ao

[11] Wheeler, D. A. (n.d.). *SLOCCount*. https://dwheeler.com/sloccount/

[12] Willenbring, J. M. (2019). The Layers of CSE Software Sustainability. *2019 Collegeville Workshop on Sustainable Scientific Software (CW3S19)*. https://collegeville.github.io/CW3S19/WorkshopResources/WhitePapers/CSEswSustainabilityLayers.pdf

[13] Willenbring, J. M. & Shende S. (2022). Impacting Software Quality and Process Through the Extreme-Scale Scientific Software Stack (E4S) and Software Development Kit (SDK) Projects.

[14] *xSDK Web Page*. (n.d.). xSDK: Extreme-scale Scientific Software Development Kit. Retrieved 12 01, 2020, from http://xsdk.info

[15] Zhao, Y., Yang, Y., Lu, H., Zhou, Y., Song, Q., & Xu, B. (2015). An empirical analysis of package-modularization metrics: Implications for software fault-proneness. *Information and Software Technology*, *57*, 186-203. 10.1016/j.infsof.2014.09.006