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.

SAND2022-11321C

Using Complexity Metrics with Hotspot Analysis to Support Software Sustainability

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

Gursimran Singh Walia
Department of Computer Science
Augusta University
Augusta, Georgia
gwalia@augusta.edu

***Abstract*— Software sustainability is critical for Computational Science and Engineering (CSE) software. Measuring sustainability is challenging because sustainability consists of many attributes. One factor that impacts software sustainability is the complexity of the source code. This paper introduces an approach for utilizing complexity data, with a focus on hotspots of and changes in complexity, to assist developers in performing code reviews and inform project teams about longer-term changes in sustainability and maintainability from the perspective of cyclomatic complexity. We present an analysis of data associated with four real-world pull requests to demonstrate how the metrics may help guide and inform the code review process and how the data can be used to measure changes in complexity over time.**

I. INTRODUCTION

Software sustainability is critical in the Computational Science and Engineering (CSE) domain [4]. Prior work lists several software sustainability attributes: extensibility, interoperability, maintainability, portability, reusability, scalability, and usability [8]. A complex code base is detrimental to software maintainability [1] and software sustainability.

While sometimes complex code is necessary, for example in a performance-critical part of a CSE code, it should at a minimum be well designed and clearly documented. In other words, it is essential to actively manage, not simply limit code complexity. This paper provides a practical guide to using complexity metrics to guide and inform the code review process, as well as help code teams measure and understand code complexity changes over time. We use four pull requests (PRs) from real-world CSE projects of varying sizes to illustrate the many ways the metrics can help a reviewer focus her efforts and better understand the impact a pull request would have on the codebase.

II. MOTIVATION

Code reviews are a valuable part of the software development lifecycle. Poorly reviewed code leads to decreases software quality [6]. The process for conducting code reviews, as well as the thoroughness and quality of code reviews are often inadequate for code projects in the CSE domain. Inadequate code reviews lead to less readable and understandable code, which increases the maintenance burden [2].

In our previous work [10] we discussed a variety of metrics in the context of software sustainability, including different measures of size, the number of contributors, complexity, and the Metrix++ maintenance index, which is calculated based on the size and complexity of the code base. We also briefly explored the hotspot analysis features offered in Metrix++ version 1.7.0 [7] and noted that these features can be helpful in the context of performing code reviews.

The basic hotspot feature can be used to identify regions of code with a supported metric value at or above a user-specified threshold. Metrix++ code regions include classes, functions, structs, namespaces, etc. Advanced features identify regions at or above the threshold impacted or "touched" by changes between two versions of a code, and regions of the code at or above the threshold that experienced an increase in the metric value between two code snapshots. We considered these features in the context of measuring cyclomatic complexity.

While cyclomatic complexity does not capture all aspects of maintainability, by definition, it does reflect the number of paths through the code. Because cyclomatic complexity is not a perfect measure of the complexity of a piece of code, it doesn't make sense to try to limit it blindly. Still, it can be helpful to quickly

**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. SAND2022-XXXX C*

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.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

identify trends and point out regions of code that should receive additional consideration. If cyclomatic complexity increases over time, it can increase the maintenance burden. In this paper we offer an analysis of how to effectively use the metrics presented to inform code reviews and measure long-term changes in the complexity of a code base.

The basis for our investigation was the observation that code regions of higher complexity might require special consideration during the code review process. In particular, code changes that introduce significant complexity should be analyzed to see if a simpler implementation would be preferable. If not, complex code regions should be minimally understandable to a reviewer. This understandability may be due to a good design, useful comments, or both. Complex code that is not understandable to an expert reviewer is a potential maintenance and sustainability concern. By identifying code regions of high complexity, and especially areas where complexity would increase after a proposed code change, we believe we can support code maintainability and sustainability for CSE software, and help code teams better measure and understand complexity changes in their code base.

III. APPROACH

Our hotspot analysis strategy was to consider four pull requests, two from each of two projects chosen from among the projects used in our previous work [10]. Both projects are members of the Extreme-scale Scientific Software Development Kit (xSDK). The xSDK [11] project was created to improve the interoperability and sustainability of scientific libraries that are common dependencies for scientific software. The projects are also currently funded in part under Math Libraries within the Exascale Computing Project's (ECP) Software Technologies (ST) thrust [5]. The PRs chosen are significantly different sizes to represent the types of pull requests that reviewers might be asked to review. Pull requests primarily or exclusively changing build infrastructure or documentation were not considered because the tool measures only the complexity of source code.

Below is a brief description of the two software projects chosen for our metric collection activity. The project names have been changed to guard against unintended conclusions concerning the sustainability of any specific project.

Project 1 includes linear and non-linear solvers and preconditioners for partial differential equation-based systems of equations. Project 1 is written primarily in C and contains more than 800,000 lines.

Project 2 is a collection of solvers and enabling technologies used for large-scale, complex multi-physics engineering and scientific problems. Project 2 is written primarily in C++ and contains over 4,000,000 lines.

For each pull request, we gathered several pieces of data. First was the number of lines added and deleted by the pull request, as provided by GitHub and GitLab. (GitLab uses the term merge request instead of pull request, but for simplicity, we use pull request or PR throughout the paper.) Note that a modified code line counts as one line added and one deleted. Second, for each of the three thresholds chosen per project, we used the Metrix++ hotspot feature to measure

1) The number of regions of code throughout the project with a complexity equal to or greater than the threshold value after applying the pull request to the code base.
2) The number of code regions throughout the project with a complexity equal to or greater than the threshold value before applying the pull request to the code base.
3) The number of regions of code throughout the project that were “touched” (modified) by the pull request with a complexity equal to or greater than the threshold value after the PR is applied.
4) The number of touched regions of code with a complexity equal to or greater than the threshold value after the PR is applied for which the value increased due to the changes in the pull request.

After generating database files for the git SHAs denoting the state of the code before and after each pull request, the data for 1 and 2 above was gathered using commands of the form:

metrix++ limit --db-file=proj1.after.complex.db
--max-limit=std.code.complexity:cyclomatic:50

The data for item 3 was gathered using commands of the form:

metrix++ limit --db-file=proj1.after.complex.db
--db-file-prev=proj1.before.complex.db
--max-limit=std.code.complexity:cyclomatic:50
--warn-mode=touched

The data for item 4 was gathered by changing the warn-mode option in the above command to “trend” instead of “touched.” Note the touched and trend metrics require the specification of a database file based on the previous state of the code (using *--db-file-prev*) to compare against.

The thresholds chosen for each project were roughly ten times (very high complexity), two times (high complexity), and one time the average complexity of regions within the code. The average complexity for Project 1 as computed by Metrix++ is 4.58. The thresholds used for Project 1 were 50, 10, and 5. The average complexity for Project 2 was calculated to be 2.30 and the thresholds were 25, 6, and 3.

These threshold levels were chosen because each may represent a different level of interest for reviewers. Changes near the current average complexity may not warrant intense scrutiny if the team is satisfied with the current complexity

of the code, but might be of greater interest for a team trying to significantly improve existing code. Regions of code with a complexity greater than two or especially ten times the average complexity should be considered more carefully.

That said, the basis for and number of thresholds should be chosen considering project and organization preferences and goals. For example, one team might find it useful to utilize five or ten different thresholds, while others may find that many to be more distracting than practically useful.

IV. RESULTS

Tables 1-4 contain the results we gathered for the four pull requests described above. The data can be understood as follows. For the first pull request, featured in Table 1, the number of lines added or modified is 112. The number of lines removed or modified is 40. This means that if there were 30 lines of modified code, ten were removed and 82 were modified. (Note that for monitoring the change in the size of the code base over a longer period, it may be additionally helpful to consider using a tool like SLOCCount [9] to see the net change in lines of code, rather than the above metrics, which blend lines added and modified into a single number.) Before applying the PR to the code base there were 4067 regions of code with a cyclomatic complexity of at least 5. After applying the PR, the number of such areas increased to 4069. Four regions were modified or "touched" by the PR. Of those, the cyclomatic complexity of 3 of the regions increased.

TABLE 1: Hotspot Complexity Data for Pull Request 1

Project 1	Lines + 112		Lines - 40	
Threshold	After	Before	Touched	Trend
50	155	155	1	1
10	2117	2117	2	1
5	4069	4067	4	3

TABLE 2: Hotspot Complexity Data for Pull Request 2

Project 1	Lines + 5279		Lines - 448	
Threshold	After	Before	Touched	Trend
50	157	155	4	2
10	2132	2118	20	15
5	4089	4070	28	20

TABLE 3: Hotspot Complexity Data for Pull Request 3

Project 2	Lines + 8		Lines - 3	
Threshold	After	Before	Touched	Trend
25	2420	2420	0	0
6	14184	14184	1	0
3	24996	24996	1	0

TABLE 4: Hotspot Complexity Data for Pull Request 4

Project 2	Lines + 1339		Lines - 668	
Threshold	After	Before	Touched	Trend
25	2362	2362	4	1
6	13071	13069	37	10
3	22614	22613	57	11

V. USING THE RESULTS DURING CODE REVIEWS

Regarding how the results can be used during code reviews, we will begin by looking at how a reviewer may use the data from Table 1 as part of a code review process. First, the line addition and removal statistics inform the reviewer about the pull request size. This information is available from both GitHub and GitLab and it provides additional context for interpreting the Metrix++ metrics.

On average, more minor pull requests in terms of lines added should be expected to introduce less complexity than larger pull requests. A more significant pull request will introduce some complexity, which should be viewed proportionally to the size of the PR. A pull request that removes many lines of code, for example, a PR that removes large blocks of previously deprecated code, can potential decrease the code's complexity metrics. A refactoring PR may cause an increase or decrease in complexity.

After the size metrics, we consider the metrics counting the number of code regions above each threshold before and after the PR is applied to the code base. These metrics can be used to understand, at a high level, the impact on the PR's complexity. The before and after counts are the only metrics collected that can help to see if the number of regions above the threshold complexity decreased. This is because areas of code with a complexity level above one of the thresholds before the changes, but below after the changes will not appear in the touched or trend metrics.

At a glance it is clear how many additional regions of complexity at or above the threshold are added by the PR. If this number is high relative to the size of the commit, it may indicate that the implementation of the features in the PR should be scrutinized to see if a simpler approach would be possible. It is also a reminder to make sure that the changes are understandable as code with a high complexity that is not written in an understandable way, or does not have sufficient documentation, is often a maintenance challenge.

The final metrics to consider are the touched and trend metrics. First, the number of touched regions provides a sense of the general complexity of the code near the changes. It might be that the changes are not introducing a lot of new complexity. Still, if the changes are touching complex code, it represents an opportunity for the PR to

improve the understandability of the code through refactoring, improved design, documentation, etc. Such improvements would be consistent with a team that adopts the legacy code change algorithm [3]. If the PR is not touching many higher complexity regions of the code, even if the code base overall is complex, complexity may not be a significant concern for the PR in question.

Finally, the number of regions exhibiting an upward trend above the given thresholds provides excellent insight into the amount of complexity introduced by the pull request. Introducing complexity may be necessary for a particular situation, either for functionality or performance. Still, ensuring that the complexity is understandable to the reviewer in those cases is crucial. When considering the very high complexity code regions (those above the highest of the three thresholds) that exhibit an upward trend in complexity, we recommend reviewing the Metrix++ output for each of these regions individually.

Example output for an upward trending region is shown in Figure 1. This output provides a couple of pieces of helpful information. First is the region name. We recommend "tapping the brake" when reviewing code in regions of very high complexity with an upward complexity trend. A reviewer should take special note of these regions when conducting the review and should minimally consider if the added complexity is necessary and if it is if the code in the region is understandable. This can also be done for code regions of very high complexity that are only touched, but do not trend upward in complexity. The second piece of important information from Figure 1 is the Change trend. In Figure 1, the complexity of the code region nearly doubled from 39 to 74. More significant Change trends mean more complexity was introduced. A very complex code region with a small Change trend indicates that the PR introduces only a small amount of additional complexity to an already complex region.

```
310: warning: Metric 'std.code.complexity:cyclomatic' for region
'buildEntityMaps' exceeds the limit.

Metric name : std.code.complexity:cyclomatic

Region name : buildEntityMaps

Metric value : 74

Modified  : True

Change trend : +35

Limit  : 25.0

Suppressed  : False
```

Figure 1: Metrix++ hotspot trend feature example output

For all the complexity metrics in Section IV, it is important to consider the size of the pull request. The pull requests considered in Section IV are of four significantly different sizes.

VI. DISCUSSION OF RESULTS

We discuss significant results concerning metrics reported above focused on key themes and contributions to understanding how the results may contribute to improving CSE software sustainability. We focus primarily on how the data can benefit code reviewers, and secondarily on how the data may be of use in a longer-term analysis of code sustainability.

Table 1 considers a pull request that adds or changes 112 lines and removes or changes 40 lines. The PR is not trivially small but is also very manageable to review. Our metrics show that the number of regions at or above our threshold complexity increases by 2 for only the smallest complexity threshold. Further, the total number of regions touched or trending up in complexity is only 4. One of these regions is a very high complexity region that is trending up. This region should receive careful consideration during the code review. The other areas can also be considered during the review based on team objectives and reviewer preference. For a PR of this size, specifically one for which it is feasible thoroughly review of all changes, one of the most significant advantages of these metrics is ensuring that added complexity does not go unnoticed.

The data for Table 2 is based on a PR that impacts more than 5000 lines of code. Depending on the nature of the changes, PRs of this size can be challenging to review. The metrics for Table 2 can help to navigate the changes in this large PR. Still, it is advisable to break larger PRs into smaller incremental pieces in many cases. If the hotspot metrics indicate many regions of increased complexity, those metrics may be used to support a request to break the PR into more manageable pieces for review.

If proceeding to review the PR in Table 2 as-is, the most critical regions of code to note are the two regions of very high complexity that are trending up in complexity. It is also worthwhile noting how much the complexity in these regions is increasing. In this case, these two regions are newly created. That can be determined based on the fact that there are two new regions of very high complexity, and precisely two regions of very high complexity are trending up. The other two touched regions cannot be the two new regions of very high complexity because regions not trending up cannot have reached the very high complexity threshold without increasing. These two newly created regions with a cyclomatic complexity of 50 or greater deserve special consideration during the code review. Knowing that there are two regions among more than 5000 lines of changes before starting a code review is helpful,

especially if the reviewer is not deeply familiar with the code or its design.

Beyond those two regions, the metrics point to other areas of possible consideration. There are two regions of very high complexity that the PR touches but does not increase the complexity of. These could be looked at to see if there is an opportunity to do some refactoring, or minimally make sure those complex regions are understandable. The next regions to consider would be the high complexity regions with an upward complexity trend, particularly those that trended significantly upward. Exactly how many of the 28 regions identified by the metrics deserve special consideration will depend on the reviewer and the policies of the software team.

The pull request associated with Table 3 is a very small pull request, adding or modifying only eight lines and removing or modifying only three. No regions of code are added above any thresholds, none within the thresholds trend up in complexity, and only one region of high complexity is touched. In the case of a small PR such as this, the most valuable role for the metrics might be to confirm that complexity was not added to the code base, and to indicate if a high complexity region of code was touched, which might indicate an opportunity to do some refactoring or improve understandability.

The data for the fourth PR in Table 4 is somewhat similar to the second PR. The PR is the second largest of the four, although it adds and modifies only about 1/4 as many lines as the second PR and removes or modifies about 50% more lines than PR 2. Based on this and the minimal number of code regions at or above threshold values before and after the changes, it seems that the fourth PR is more focused on code modification and the second on adding new code. Per line of code added, removed, or modified, the number of regions of code touched or trending up in complexity for the 4th PR is much higher; however, that PR modifies several regions of code, rather than primarily adding regions of code, and the complexity thresholds are lower for project 2, and thus easier to reach.

For this PR, it would make sense to focus attention on the very high complexity region of code that trended up in complexity, as well as consider the other three touched areas of high complexity and potentially give some consideration to the regions of code above the two lower thresholds that trended up in complexity.

While there is no concrete set of rules for utilizing the metrics in Tables 1-4, the metrics can be used to better understand the impact of pull requests and help guide the reviewer's attention during the PR review process.

The metrics included in this study can also be used in attaining a longer-term view of software sustainability. While complexity does not provide a complete picture of software sustainability for a code project, trends in complexity offer one aspect of sustainability that can be combined with others for a more holistic analysis.

To illustrate this more clearly, consider the before and after metrics for Project 2 in Tables 3 and 4. The two pull requests analyzed for Project 2 took place about 25 months apart. Over 2000 regions of average complexity, 1000 regions of high complexity and 50 regions of very high complexity were added to the code between the two PRs. Whether or not that is a lot or an acceptable amount depends on several other factors. How actively was the code being developed? This could be measured by a combination of the number of PRs or commits merged, or the change in the size of the code base. What is the ratio of lines of code to the number of very high/high/average complexity regions of code?

While we would not recommend using this data in isolation to determine if a code base is becoming more or less maintainable or sustainable, this information can be used in multiple practical ways. These include monitoring code complexity over time and helping to evaluate the effectiveness of a refactoring effort. Such metrics can also be used to help train and coach individual developers if it is determined that PRs from particular individuals tend to introduce disproportionate amounts of complexity into the code base.

VII. RELEVANCE TO INDUSTRY

The metrics and related discussion presented in this paper have exciting potential for use in industry. As outlined above, this data can be used to guide and improve PR review efforts. The hotspot metrics measure how much complexity is introduced by a pull request, how much complexity is present in the regions of code associated with a pull request and where the regions of most significant concern are. The data can be used to identify regions of code that should be refactored while working on surrounding code, and support requests that PRs be broken into smaller pieces.

The metrics and discussion can also be used for monitoring changes in complexity over time, and for helping identify individual developers who might benefit from training in code design. We have provided an analysis of how these metrics can be used, but we recommend customizing the approach to the organization's or project's needs. These needs determine how aggressively to limit new or reduce existing complexity. Another critical area of customization is in choosing complexity thresholds. Our examples used three thresholds, and set the threshold values to roughly one, two, and ten times the project's average code region complexity. Based on experience, teams may choose

different thresholds, or an organization might set different complexity targets across projects.

The data collection for our analysis is easy to automate using a continuous integration capability such as GitHub Actions or GitLab CI. The process would simply need access to Metrix++, and then the git SHAs associated with the current state of the branch the PR is being proposed to be added to, as well as the git SHA for the state of the branch that includes the PR.

Given the usefulness of the metrics collected for the code review process, the longer-term analysis of code maintainability and sustainability, and the fact that the collection of the needed data can be easily automated, the approach described above has excellent potential to impact the sustainability of CSE software positively.

VIII. FUTURE WORK AND CONCLUSION

In the future, we plan to study more carefully pull requests focused primarily on code removal and cleanup in addition to our current efforts to look at code additions and modifications. We also plan to gather more data on tracking changes in complexity over time, while concurrently looking at changes in the overall codebase size. The Metrix++ maintenance index metric can be used for this analysis. A Jupyter Notebook might be the right medium for presenting this information.

Another area of interest is to look at the complexity metrics on a per-contributor level, which may help to identify opportunities to train and coach software developers. We may also consider different approaches for choosing complexity thresholds for our analysis. A final area of future work would be to automate the collection of complexity metrics and make the information available to code reviewers, which would allow us to quantify the metric's utility more precisely.

We analyzed pull requests from two representative code projects that are part of the xSDK and the US DOE Exascale Computing Project. The findings and analysis based on this real-world data can be generalized beyond the CSE domain as the data simply reflected the amount of complexity existing in the code before a pull request as well as the amount of complexity that would be added or removed by the pull request. While this data varies from PR to PR and project to project, measuring and understanding complexity is helpful regardless of whether the amounts of existing and newly proposed complexity is high or low.

Highly complex code makes software more difficult to maintain, and less sustainable. However, writing complex code is sometimes necessary due to feature or performance requirements. For these reasons it is essential to manage complexity, not simply limit it. The ideas in this paper can help code teams be very intentional about managing complexity in the codebase. Specifically, the metrics and techniques presented provide opportunities to eliminate complexity at code review time before the complexity enters the codebase or alternatively ensure that complexity is well documented and well designed to minimize the negative impact on maintainability. Further, the metrics point to regions of existing complexity that a team may choose to refactor when modifying nearby code. By managing complexity on a PR-by-PR basis and monitoring complexity trends over time, a code team can improve the sustainability of their code base.

Bibliography

[1] Boehm, B. Software Engineering Economics, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[2] Eisty, N.U., Carver, J.C. Developers perception of peer code review in research software development. *Empir Software Eng* **27,** 13 (2022). https://doi.org/10.1007/s10664-021-10053-x

[3] Feathers, M. Working Effectively with Legacy Code. Prentice Hall, 2005.

[4] 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*. https://www.nitrd.gov/PUBS/CSESSPWorkshopReport.pdf

[5] 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

[6] McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2014). The impact of code review coverage and code review participation on Software quality: A case study of the Qt, VTK, and ITK projects. In *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings* (pp. 192-201). Association for Computing Machinery, Inc. https://doi.org/10.1145/2597073.2597076

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

[8] 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

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

[10] Willenbring, J., Walia, G. Evaluating the Sustainability of Computational Science and Engineering Software: Empirical Observations. In *34th International Conference on Software Engineering & Knowledge Engineering, SEKE 2022 – Proceedings* (pp 453-456). KSI Research.

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