eCOMMONS

Loyola University Chicago

Loyola eCommons

Master's Theses

Theses and Dissertations

2016

Metrics Dashboard Services: A Framework for Analyzing Free/Open Source Team Repositories

Fnu Shilpika
Loyola University Chicago

Follow this and additional works at: https://ecommons.luc.edu/luc_theses

Part of the Computer Sciences Commons

Recommended Citation

Shilpika, Fnu, "Metrics Dashboard Services: A Framework for Analyzing Free/Open Source Team Repositories" (2016). *Master's Theses*. 3270.
https://ecommons.luc.edu/luc_theses/3270

This Thesis is brought to you for free and open access by the Theses and Dissertations at Loyola eCommons. It has been accepted for inclusion in Master's Theses by an authorized administrator of Loyola eCommons. For more information, please contact ecommons@luc.edu.

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License.
Copyright © 2016 Fnu Shilpika

LOYOLA UNIVERSITY CHICAGO

METRICS DASHBOARD SERVICES:
A FRAMEWORK FOR ANALYZING FREE/OPEN SOURCE TEAM REPOSITORIES

A THESIS SUBMITTED TO

THE FACULTY OF THE GRADUATE SCHOOL

IN CANDIDACY FOR THE DEGREE OF

MASTER OF SCIENCE

PROGRAM IN COMPUTER SCIENCE

BY

SHILPIKA

CHICAGO, ILLINOIS

AUGUST 2016

Copyright by Shilpika, 2016
All rights reserved.

ACKNOWLEDGMENTS

I would like to thank everyone who made this thesis possible, starting with my wonderful professors in the Computer Science department at Loyola University Chicago. Dr. Nicholas J. Hayward provided excellent visualizations of the Metrics Dashboard service results; the graphs used in the thesis for the evaluation of the Metrics Dashboard API were generated by the front end service that was developed by him. Prof. Konstantin Läufer provided guidance when I ran into any type of coding complications. Dr. Venkatram Vishwanath of Argonne National Laboratory granted us access to the supercomputing resources that helped us perform computations on larger projects. Finally, I would like to thank my thesis committee director and advisor, Prof. George K. Thiruvathukal. He proved to be an excellent mentor from the beginning of my time here at Loyola University and steered me towards thinking about the best practices in Software Engineering. Without his help this work wouldn't have been possible. His friendship and encouragement have made the difference in this arduous process.

I would also like to thank the National Science Foundation (NSF) for funding our research. A fifteen month Research Assistantship during 2015-2016 allowed me to make discernible progress in our research.

My friends all over the world have provided me with the much needed cheering and distractions after which I was able to continue my work with a refreshed mindset.

Finally, I would like the love of my life, my husband and my best friend, Nischith Chandrashekar for believing in me. I would never have made it to where I am without his love and support.

To my parents and my husband.
Thank you for your support and love.

PREFACE

Progress in scientific research is dependent on the quality and accessibility of software at all levels. True progress in software development depends on embracing the best traditional--and emergent-- practices in software engineering, especially agile practices that intersect with the tradition of software engineering [1]. Measuring software quality can lead to developers following good software engineering practices. Software processes can use the best features and practices of various models which is suitable for that project. To identify these features and practices it becomes necessary to measure software quality. Measurement, in essence, captures information about the attributes of an entity being measured. When it comes to software measurement, it becomes essential to identify these attributes that would eventually contribute towards providing meaningful (although not complete) information about a software product. This could lead the embracement of best practices that is important to develop and maintain good reusable software. In this thesis, we aim to identify software metrics derived from commonly used metrics like defect count and lines of code; we then implement these derived metrics and provide a dashboard view to the software teams which would give them an outline of how the software development is progressing. With this work, we hope to lay the groundwork for using software metrics to identify software engineering problems and come up with software engineering practices to fix them.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

PREFACE vi

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT x

CHAPTER ONE: INTRODUCTION 1
Broader Context 3
Related Work 4

CHAPTER TWO: DESIGN AND IMPLEMENTATION 8
Metrics Dashboard Functionality 8
Longitudinal metrics and sampling 8
Module size computation 9
Supported metrics 11
Metrics Dashboard 16
Architectural overview 16
Analysing sample visualisations 19

CHAPTER THREE: CONCLUSION 28
Evaluation 28

LIST OF REFERENCES 31

VITA 33

LIST OF TABLES

Table 1. Open source projects tracked by default as of May, 2016 18

LIST OF FIGURES

Figure 1. Commit history flow diagram for a file (astropy/table/column.py) 10

Figure 2. Architectural overview of the metrics dashboard service 16

Figure 3. Go: Line chart for density and KLOC against month 19

Figure 4. Go: Line chart for density and spoilage against month 23

Figure 5. Go: Line chart for issues grouped by week 24

Figure 6. Sympy: Line chart for issue density and KLOC against month 25

Figure 7. Sympy: Line chart for issues grouped by week 26

Figure 8. Sympy: Line chart for issue density and spoilage against month 26

ABSTRACT

There is an emerging consensus in the community that "progress in scientific research is dependent on the quality and accessibility of software at all levels" [1]. This progress depends on embracing the best traditional---and emergent---practices in software engineering, especially agile practices that intersect with the more formal tradition of software engineering. As a first step in our larger exploratory project to study in-process quality metrics for software development projects in Computational Science and Engineering (CSE), we have developed the *Metrics Dashboard*, a working platform for producing and observing metrics by mining open-source software repositories on GitHub. The Metrics Dashboard allows the user to submit the URL of a hosted repository for batch analysis, whose results are cached. Upon completion, the user can interactively study various metrics over time (at different granularity), numerically and visually. We currently support project size (KLOC), defect density, defect spoilage, and productivity. The Metrics Dashboard distinguishes itself in various ways: 1) it is free/open-source software distributed under a license still to be determined; 2) it has an extensible architecture that makes it easy to study additional metrics; 3) it provides both a human-facing web application and a RESTful web service for consumption by programmatic clients; 4) it is hosted as a publicly available software-as-a-service (SaaS) instance, and users and contributors can choose to self-host their own instance; and 5) batch processing. We have implemented the Metrics Dashboard using modern web

service/application technologies and a scalable architecture. While this work is part of an effort to address sustainable practices in scientific software development, we believe it to be more broadly applicable to any interdisciplinary software development community.

CHAPTER ONE

INTRODUCTION

Software engineering as practiced today (especially in the industry) is no longer about the stereotypical monolithic life cycle processes (e.g. waterfall, spiral, etc.) found in most software engineering textbooks (aimed at large scale software teams). These heavyweight methods historically have impeded progress for small or medium sized development teams owing to their inherent complexity and rather limited data collection strategies that predominated the 1980s (a fervent period for emerging software engineering research) until relatively recently in the mid-2000s. In addition, the discipline and practice of software engineering includes software quality, which has an established theoretical foundation for doing software metrics (a fancy word for measuring). In our work, we try to explain how software processes can be pragmatic and use best features or practices of various models without impeding developer productivity, especially with a growing number of cloud-based solutions for hosting projects (the most famous being GitHub). The challenge is to cherry-pick the most-effective practices from a large suite of tools and incorporate them into existing cloud-based workflows. Development teams are particularly likely to resist using a practice if it incurs any additional workload on already short-staffed teams, especially if the solution is not integrated into existing infrastructure (e.g. GitHub, used by many open-source, computational science projects alike). The tools

we are developing can be incorporated in any existing GitHub based workflow without requiring the developers to install anything.

We begin by focusing our energy on the Metrics Dashboard, developed as a part of an NSF-funded effort for looking at Software Engineering in computational science projects. We implement traditional software metrics as described in the classic reference by Fenton [11] with a web-based dashboard so project teams can just use them to understand quality in their projects.

We provide a brief overview of what practices from software engineering can be helpful to open source and computational science and engineering projects. Many projects already use version control. But what else can they be using that would be helpful to improve software quality? Issue tracking is one that can have huge impact on projects, not only for tracking code but also for textual content.

We provide a brief overview of (software) quality: Many of us know the English definition of quality, but this definition differs from the one created by W. Edwards Deming--an exponent of quality (in manufacturing) who focused his definition on customer expectations being met or exceeded. Customer expectations (a.k.a. satisfaction) is a key driver of process improvement (the idea being that you cannot improve something you don't understand; measurement is a key to establishing an understanding of any process).

We provide an overview of software metrics, focused on so-called in-process metrics (as opposed to code-based metrics, which are also useful but not the scope of work).

We look at two specific (and challenging) software metrics (defect density and spoilage) for 10 active open source projects on GitHub that employ software practices to help us compute the metrics accurately (git commits and issue tracking). The Metrics Dashboard effort itself is built using agile software development methods.

Lastly, we analyze all of the GitHub projects history (git and issues, among others) and the process is highly data intensive. Our current focus is only on 10s of projects but will be scaled to all-known computational science projects in 2016.

Broader Context

Software metrics are a critical tool that provide continuous insight to products and processes and help build reliable software in mission-critical environments. Using software metrics we can perform calculations that help assess the effectiveness of the underlying software or process. The two broad categories of metrics are:

1. Structural metrics, which tend to focus on intrinsic code properties like code complexity.
2. In-process metrics, which focus on a higher-level view of software quality, measuring information that can provide insight into the underlying software development process.

We understand that metrics are often used to evaluate individual developer productivity rather than overall project quality and progress. For example, a large number of commits made to a project may or may not have any impact on the software quality (yet it is displayed prominently on sites like GitHub). Optimizing on one metric could result in unintended consequences for a project. For example, these commits could be

overly complex or introduce defects. Therefore, we seek to identify metrics that will be useful to the project as a whole.

Our aim is to develop and evaluate a Metrics Dashboard to support Computational Science and Engineering (CSE) software development projects. This task requires us to perform the following activities:

1. Assess how metrics are used and which general classes or types of metrics will be useful in CSE projects.
2. Develop a Metrics Dashboard that will work for teams using sites like GitHub, Bitbucket etc.
3. Assess the effectiveness of the Metrics Dashboard in terms of project success and developer attitude towards metrics and process.

Our current focus is on identifying requirements for the Metrics Dashboard, which include the types of metrics that will help understand and improve the software quality.

Related Work

During the design and implementation of the Metrics Dashboard, we have relied on methods and insights from the mature yet dynamic field of *software architecture*. By unifying a body of independent prior work, the seminal report by Garlan and Shaw [2] defines software architecture as a design perspective that focuses on the overall module structure of increasingly complex software systems (as opposed to details of data structures and algorithms); the report surveys common architectural styles and compares their effectiveness and responsiveness to change based on several case studies. Updated,

comprehensive studies of this subject are available as well [3]. During the last two decades, it has become increasingly common to design distributed systems that provide common functionality by combining separate applications and services. This trend has drawn attention to the field of *(enterprise) application integration*, where common integration styles include file transfer, shared databases or repositories, remote procedure invocation, and messaging [4]. The research on comparison between the metrics identified is similar to the comparison of defect density and change density by [5]. The work done by Shah, Morisio and Torchiano [6] studied 19 papers that reported defect density for 109 software projects and found that larger projects exhibit lower defect density than medium and small projects. They have compared defect density for the programming language, Java, C++ and C and state that the difference in defect density could be attributed to different level of detail and expressive power between the two languages. In [6] the analysis of size, age, programming language and development mode of project (close vs. open) could be factors for defect density was tested for and it was found that development mode is a factor with programming language affecting the values in some cases. In addition it was found that projects size is relevant, while age was not a factor.

In the study by Gala et al. [7] the ratio of the email messages in public mailing lists to versioning system commits which has remained constant along the history of the Apache Software Foundation (ASF), was found to be independent of the size, activity and number of developers and relatively independent of the technology and functional area of the project but seems to be technical effervescence and popularity of the project.

They have studied ratio of developer message to commits and ratio of issue tracker message to commits. The metrics identify stagnant projects or projects in the verge of stagnation. They is still verification pending to see if these results apply to other open source projects and if these results are practical. Defect density is a high level metric which may lead to different interpretations so two different variants of this metric are used, in the work by Shah et al. [8] standard(steadily increasing variability) and differential(large variability). The conclusion here is that the standard defect density provides a global (with all history included) quality view of the project and differential defect density provides a local (specific to a version) quality view to a project. Differential defect density varies between 1 - 100 defects per KLOC which could be attributed to defects between releases which belongs to the previous release. The steady growth of standard defect density means either that the quality of a project decreases over time, or that this metric is not a reliable quality indicator. As for differential density, its high variability could be either normal behavior, or an indicator of a project that is not under control. In the latter case, projects should try to reduce differential defect density as much as possible. The work done by Nagappan and Ball [9] determines if the defect density identified by static analysis tools like PREfix and PREfast help predict the pre-release defect density i.e. the defect density identified by developers. In order to address the fact that the results were not coincidental they repeated the data splitting experiment several times and provided consistent results each time. The static analysis tool used in this paper mainly works with C and C# modules to identify defects and calculate the defect density. This highly limits its usage to projects that use C and C#. The static tools

sometimes detect false positives that could identify modules as error prone even if they are not as error prone as reported. Static Analysis tools are known to miss deep functional and design errors which are normally caught by programmers while testing, so this type of automated testing is less efficient than manual testing by programmers.

The paper by Bower et al. [10] proposes a catalog-driven approach to look at qualitative and quantitative dimensions of software metrics. This approach is needed because there are so many metrics, and there is little or no attempt to collect (in one place) which are effective and in what situations. Most approaches are ad hoc and provide little structure. This paper is an attempt to bring structure to the forefront by having a sound cataloging scheme. There are many surveys for certain metrics (e.g. object-oriented, etc.) but most discussions about them are qualitative, so there is little way of knowing which work. This paper is taking some baby steps toward addressing this issue. There is a table of qualitative aspects and quantitative aspects. A particularly interesting aspect of quantitative is to distinguish between base metrics vs. derived metrics, which has come up in our discussions. In this thesis, we work with derived metrics. The vocabulary in the table could be helpful to us for thinking more deeply about the metrics we are implementing and understanding their long-term value in actual projects during the evaluation phase of our work.

CHAPTER TWO

DESIGN AND IMPLEMENTATION

Metrics Dashboard Functionality

Longitudinal metrics and sampling

Any process, including the software development process, occurs over time and is therefore longitudinal in nature. In this study, we aim to study the development process through metrics that must themselves be longitudinal, that is, they are functions of time. For example, code size (KLOC) may change over time whenever a committer inserts or deletes portions of the code. While these metrics are conceptually continuous functions of time, it is impractical to treat them as such, and one typically uses sampling to convert them to discrete functions of time. The choice of sampling rate (frequency) is a practical one. If one wanted to observe, say, intra-day phenomena, one would choose a relatively high sampling rate, such as hourly or even every 15 minutes. Our study, however, focuses on longer-term phenomena, so the commonly used daily sampling rate will be sufficient. In practice, daily measurements are taken at midnight local time (00 hours). Less frequent samples can always be obtained by downsampling (decimation) as follows. The measurement for a metric $y(d)$ for a calendar interval $[d_0;d_1]$, where d_i are calendar dates, is given as the daily average of y over the time interval:

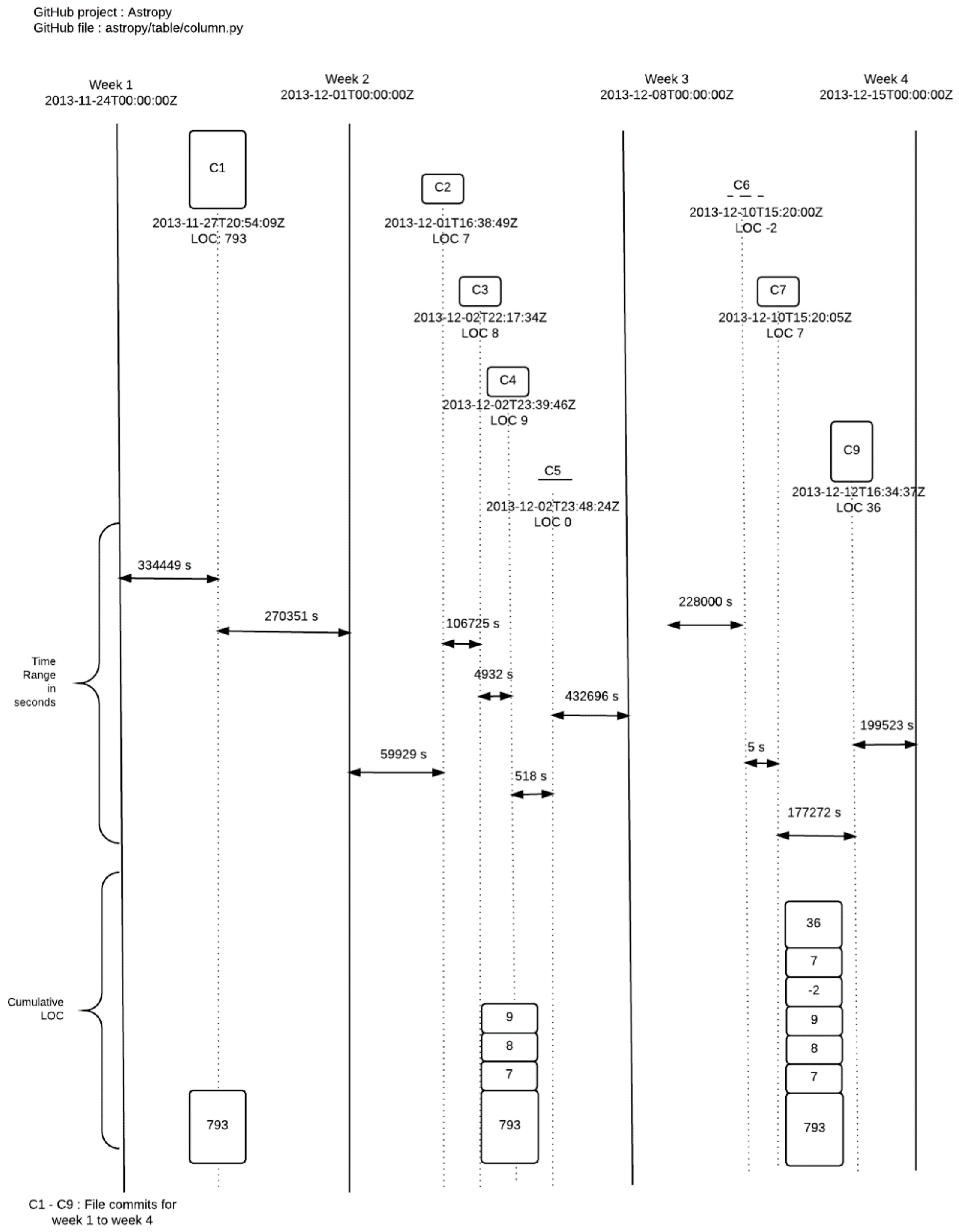
$$y([d0;d1]) = \frac{\sum_{d=d0}^{d1} y(d)}{d1 - d0} \tag{1}$$

Using this definition, we can obtain measurements by week, month, or other arbitrary period.

Module size computation

Figure 1 shows how the module size is calculated for a GitHub project named Astropy. In software engineering, the module size is calculated using LOC. A project will consist of large number of files, each file with its own commit history. In this thesis, the final metrics results are grouped with granularity of week or month, therefore the module size is calculated for the requested granularity. Keeping this in mind, the module size for the entire project is calculated by looking at the commit history of each file in the project and partitioning the commit history based on the requested granularity. Referring to figure 1, the calculations are shown for weekly granularity with the commit history for the file astropy/table/column.py starting from November 24, 2013 to December 15, 2013. The first commit for the file was made on November 27, 2013 and hence the LOC value for the dates before the first commit for the file remains zero. Week one shows one commit for the file, therefore, the LOC value for the file is 793, from the file first commit date (November 27, 2013) to the second commit date (December 1, 2013). On the second commit, the LOC of the previous commit is added to the current commit, in this case, the second commit was 7 LOC giving a cumulative commit result, after the second commit date to the third commit date, of 800 LOC. The process continues for subsequent commits on the file. A point to note here is that the LOC can contain zero or negative values, in each case we simply add these signed values to the previously calculated cumulative LOC.

Figure 1. Commit history flow diagram for a file (astropy/table/column.py)



Due to longitudinal nature of our computation we calculate the time that has elapsed between each commit in milliseconds. These results are later converted to seconds for visualization, therefore giving us a result of cumulative LOC multiplied with the time range between the current commit to the next commit. The final result for each file (time range * cumulative LOC) added with every other file in the project that falls under the same granularity (week or month) or window. The merged result (time range * cumulative LOC) is divided by the duration of the requested granularity, therefore we end up with the with the module size with the unit in LOC instead of seconds-LOC. The calculations of module size is crucial for the metrics calculations of issue density and productivity.

Supported metrics

Issue density is the number of confirmed defects detected in software/component during a defined period of development divided by the size of the software or component [11].

Defect density is usually shown as the number of reported software defects per 1,000 lines of source code (KLOC).

$$Defect\ Density\ =\ Number\ of\ Defects\ /\ Module\ Size \tag{2}$$

In this thesis, the focus is on open source projects in GitHub, therefore this metric is referred to as issue density. GitHub provides a feature for tracking tasks, enhancements and bugs for a project and is referred to as issues. Since, open-source projects in GitHub use the issue tracking feature extensively for tracking their project bugs, we use the count of issues for calculating the number of defects, in this case issues, for the computation of

issue density. For our work, we are focused on projects that use the issue tracking feature of GitHub extensively, one of the reasons being that, the issue count in GitHub gives us an idea about how active the open-source user community is for the identified projects along with how actively the contributors to the project keep track of issues. For example, how promptly the issues are closed or updated, how well the software is being tested, how promptly the outcomes of various types of tests and peer code reviews are tracked.

The module size for a given project in a repository is calculated using the KLOC (thousands of lines of code) for that project. To count the KLOC for a chosen project we count the lines of code at each commit for that file, multiply that with the duration until the next commit; this is repeated for the entire history of the file till the current date and the final result is divided by the entire range for which the file exists in GitHub. The section on module size computation explains, in detail, how we calculate the KLOC and how we arrive at the KLOC result. KLOC per file is given as follows:

$$KLOC_{(file)} = Duration_{(BetweenCommits)} * LOC/(FileCommitDuration) \quad (3)$$

The file commit duration above is the duration for which the KLOC for a project is calculated by considering all source files that belong to the project and is given by

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{m} KLOC_{(i,j)} &= \sum_{i=1}^{n}(KLOC_{(i,1)} + KLOC_{(i,2)} + \\ &\quad KLOC_{(i,3)} + ... + KLOC_{(i,m)}) \end{aligned} \quad (4)$$

where n = number of files in a repository and m = granularity requested by the client; for example, week, month.

The defect density metrics can be granulated to give the result grouped either by month or week. Instead of using the direct measurement of faults we compute the fault

density, i.e. derived measures (combination of measures) defined as issues per KLOC. The idea here is intuitively comprehend how the software development is progressing. For example, if a project has very few issues but the KLOC metric is increasing, this could either mean that the issues identified in the given window (month, week) is being closed within the window or it could be an indicator of a much larger problem of the user community or contributors for the project are not as active as they should be in testing or tracking the issues. In this way, we try to encourage better project maintenance by the users and developers while the project is being developed.

Issue spoilage refers to how much effort was spent in fixing faults rather than building. This can also incorporate the idea of cost of fault prevention compared with the cost of fault detection and correction.

Issue spoilage = effort spent fixing faults / total project effort.

$$Issuespoilage = \sum_{i=1}^{n} \frac{T_{issues(i)}}{T_{lastCommit} - T_{firstCommit}} \quad (5)$$

where n = number of issues.

We calculate the time taken to fix issues logged in GitHub. An issue is considered to be fixed when its status is changed to *close*. An issue in any other state is considered to be *open*. Our approach is to intuitively identify the project health. If the spoilage value increases over time, this could indicate the following:

1. The project issues are being neglected and not closed as quickly as they should be.

2. The project doesn't have enough contributors and the developers working on the project are overwhelmed with too much work.

On the other hand, if the spoilage values are reducing this could indicate the following:

1. The user community is not actively participating in identifying issues.
2. The project issues are being closed fairly quickly and this is a good indicator of a project that is doing well.

One can be fairly certain about the project health, when it comes to spoilage, by looking at other metrics like issue density, which gives us the issues per KLOC:

1. If the issue density value has increased for the chosen granular window and the spoilage has reduced for the same window then it is a good indicator that the project is being maintained well.
2. If the issue density value has decreased for the chosen granular window and the spoilage has reduced for the same window, it could signify that the currently active issues are being closed fairly quickly but not much effort is being expended at identifying new issues.
3. If the issue density value has increased for the chosen granular window and the spoilage has increased for the same window, this could indicate that the user community is actively identifying issue in the code base but the issues aren't being closed quickly enough.
4. If the issue density value has decreased for the chosen granular window and the spoilage has increased for the same window this is an indicator that the project

development has slowed down for that window and steps should be taken to improve these values.

Productivity is the most commonly used model for productivity measurement expresses productivity as the ratio of “process output influenced by a personnel” divided by the “personal effort or cost during the process”. Since our work is focused on measuring the productivity of a team we define productivity as follows:

$$Productivity = ModuleSize/TeamEffort \tag{6}$$

Module size is one of the measures that is used to compute productivity and is calculated as per the examples provided in the section module size computation. The module size is given by (3). The team effort is calculated by considering the development time of the project. Since the results are sampled based on the chosen frequency, the team effort is given by the time elapsed between the first commit for the project and the last commit for the project made in that window.

$$TeamEffort = T_{lastCommit} - T_{firstCommit} \tag{7}$$

This metric combines the process measure (Team Effort) and the product measure (module size). Our goal is to check how much effort is being spent in fixing issues when compared to actual code development. In order to facilitate the production of quality software any software development team should be focused on releasing code with minimal defects or in this case, code that would have minimal issues associated with it. This in turn makes sure that the less effort is expended on fixing issues and allows for more focus on software development, thereby reducing the overall development time and hence reducing the cost of development. A lower value in productivity means poor

software quality which could result from using too few people or people with the wrong skills.

Our approach with these identified metrics is to give a clearer picture of software quality. From our experience in both industry and academia, we understand that no one metric can give a clear indication of project health. Viewing the metrics results in conjunction will give us a better idea of software health and pave the way towards stronger development strategies in future deployments and releases.

Metrics Dashboard

Architectural overview

Figure 2. Architectural overview of the metrics dashboard service

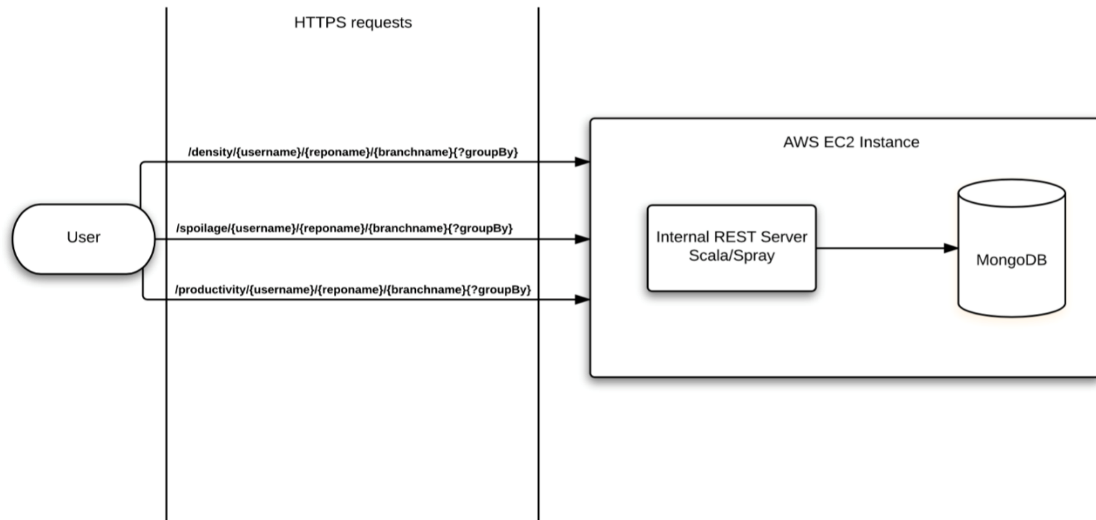


Figure 2 gives the architectural overview of the Metrics Dashboard service. The Metrics Dashboard service is currently hosted on the Ubuntu Linux instance running on the Amazon Elastic Compute Cloud (Amazon EC2) on the Amazon Web Services platform. The server side application is developed using Spray which a lightweight Scala

library providing server side and client side REST-HTTP support on top of the Akka toolkit. One can build highly concurrent, distributed and message driven applications on the JVM using the Akka toolkit. The persistence of the computed results is done via MongoDB which is a document based database and since the server side application provides a JSON result this proves to be a suitable choice. The Metrics Dashboard API can be accessed using the URL structure, *https://tirtha.loyolachicagocs.org/metrics/api/{metric-type}/{user/organization}/{repository}/{branch}?groupBy={frequency}*.

1. Metric type tells the web service what metric one is looking for. Currently, we support three types of metrics which include issue density, which is accessed using the term *density*, issue spoilage, which is accessed using the term *spoilage* and Productivity, which is accessed using the term *productivity*.
2. User or Organization refers to the username or organization name in GitHub under which the project of interest is stored.
3. Repository is a container in GitHub within which a project resides.
4. Branch is the name of the project branch that one would like to access. A GitHub branch could contain different versions of the same project.
5. The parameter groupBy in the URL is the frequency or granularity with which one would like to view the final results. Currently, we compute the result based on monthly and weekly frequencies.

The following GitHub open-source projects are being tracked by the Metrics Dashboard Service by default.

Table 1. Open source projects tracked by default as of May 2016

Project	Commits	Open Issues	Issue Density (Issues/KLOC)	Issue Spoilage	Productivity (KLOC/ms)
IPython	21,518	959	2.321	5.6609	6.2445
SymPy	25,010	2340	1.7566	17.4247	4.4758
Astropy	15,480	718	0.7850	10.5402	4.0056
Simbody	4,542	82	0.1367	0.4798	3.1635
Numpy	14,777	1211	3.3737	5.6239	6.7095
Go	28,460	2295	2.3565	3.4880	1.8716

When we take a look at a project like Simbody, one can immediately note that the number of issues for the project is 82. Thereby, the issue density (issues per LOC) is a lower value, 0.1367. This would lead us to conclude that the developers are fixing the issues that are identified fairly quickly. However, the matter of concern here is that for a project with KLOC in the range of approximately 599 *https://tirtha.loyolachicagocs.org/metrics/api/density/simbody/simbody/master?groupBy =month}* in May, 2016, the issue count seems to be lower. This could mean the team is more focused on fixing currently identified errors in their codebase rather than testing and identifying potential bugs or issues that could break the project. Productivity for the project seems to be a decent value, one way to come to this conclusion is to make sure that this metric stays above one. We want a project where effort spent is lower, so whatever the value is for a particular window of chosen granularity, that value should

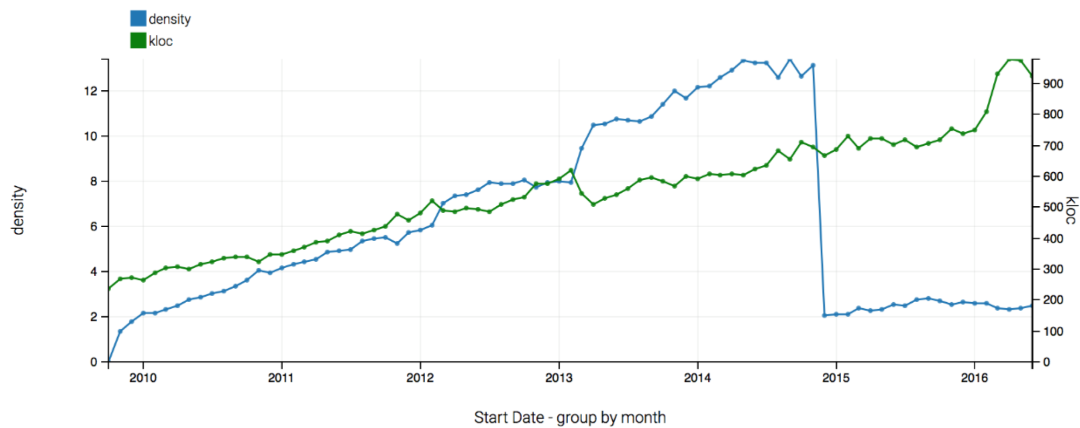
either stay the same or steps should be taken to increase the value for the next window of chosen granularity.

Analysing sample visualisations

An attempt at a brief analysis of the projects that are tracked by default is performed in this section.

Project Go is an open-source programming language developed by Google and the repository is hosted in GiHub. Figure 3 shows the issue density and KLOC for the project against month.

Figure 3. Go: Line chart for density and KLOC against month



At first glance, the steep dip in the issue density catches the eye. This dip occurs without any corresponding changes in the KLOC for the code. We can intuitively come to the conclusion that a large number of issues were closed in a very small time frame for this project. To check this assumption for correctness, we can do the following:

1. Navigate to the GitHub issues section for the project (*https://github.com/golang/go/issues?utf8=✓&q=sort%3Acreated-asc%20*).

Here, we notice that the first issue for the project was created in October, 2009.

2. Check the metrics dashboard service (*https://tirtha.loyolachicagocs.org/metrics/api/density/golang/go/master?groupBy=week*), to identify the window where the dip occurred. The results obtained from the service will be in JSON format as shown below, which contains the fields open or close and openCumulative and closeCumulative which specify the issues opened or closed in the chosen granularity and the issues that are in the open or close state in the current window of chosen granularity.


```
{{
  "start_date": "2014-11-24T00:00:00Z",
  "end_date": "2014-12-01T23:59:59Z",
  "kloc": 651.0461298714263,
  "issues": {
    "open": 30,
    "closed": 0,
    "openCumulative": 9161,
    "closedCumulative": 0
  }},{
  "start_date": "2014-12-01T00:00:00Z",
  "end_date": "2014-12-08T23:59:59Z",
```

"kloc": 653.9527515172911,
"issues": {
"open": 34,
"closed": 0,
"openCumulative": 9195,
"closedCumulative": 0
}},{
"start_date": "2014-12-08T00:00:00Z",
"end_date": "2014-12-15T23:59:59Z",
"kloc": 639.6212584045984,
"issues": {
"open": 120,
"closed": 7968,
"openCumulative": 1347,
"closedCumulative": 7968
}}}

On close inspection, we notice that the date 2014-12-08T23:59:59Z is when the dip occurs, also notice that issues closed is 0 and closedCumulative is 0 For the next window, (2014-12-15T23:59:59Z) closed and closedCumulative is 7968.

3. Navigate to GitHub issues (*https://github.com/golang/go/issues*) to check if the values reported by the metrics dashboard service is correct. If we filter

using the criteria **closed:\textless 2014-12-08**, we see that no issues were closed before this date (*https://github.com/golang/go/issues?utf8=✓&q=closed%3A%3C2014-12-08*), even though the first issue was opened in October, 2009.

4. Change the filter to **closed:\textless 2014-12-09** (https://github.com/golang/go/issues?utf8=✓&q=closed%3A%3C2014-12-09) and we will see that 7926 issues were closed. Change the date to **2014-12-15** (https://github.com/golang/go/issues?utf8=✓&q=closed%3A%3C2014-12-15) and you will see 7968 issues were closed, which matches the result obtained through the metrics dashboard service.

So to summarize, the Go programming team managed to close 7926 issues in one day, which is not considered to be a good programming practice, considering the fact that the team hadn't closed any of the identified issues since October, 2009. A point to note here is that the status of issues in GitHub can be changed, however here were are concerned only with the open and closed dates of issues and having an issue for five years is simply not excusable. For the above analysis we choose the granularity to help easily narrow down the exact date when the dip occurred.

Figure 4. Go: Line chart for density and spoilage against month

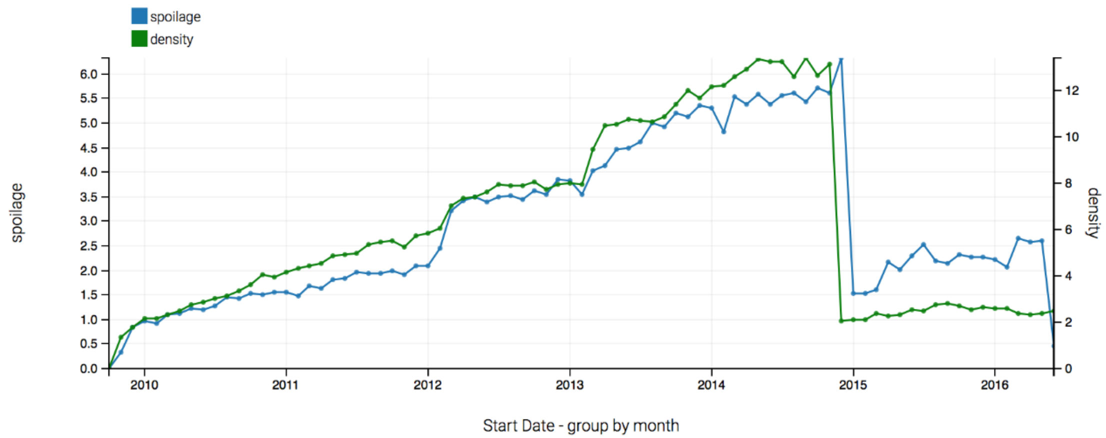


Figure 4, shows the issue density and spoilage against month, as one expects the spoilage value dips at around the same window when the issue density dips.

Another observation one can make using the visualization is that spoilage increases until the end of the year 2014 to a peak of almost 7.0, as the time to fix issues increased. After the dip the spoilage has remained constant which is a good indicator that the issues are being closed regularly and newer issues are identified and tracked. For an active project to be healthy, the spoilage should not drop too low, which could indicate that the project isn't being tested and the user community isn't actively identifying or reporting issues.

Figure 5. Go: Line chart for issues grouped by week

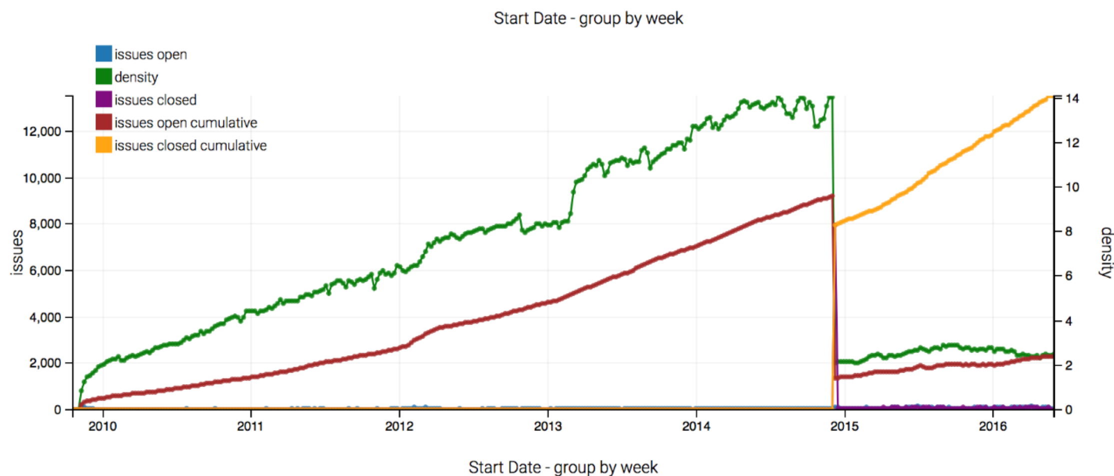


Figure 5, shows the issues for the Go project against week, as one expects the issues for the project were opened until the end of the year 2014 (cumulative open issues are shown in red). The closed cumulative issue count, shown in yellow, shows that the issues for the project were closed beginning the end of the year 2014. Since 2015, the team or users have continued to open and close issues at a fairly steady rate and no drastic changes in the values are seen. This means that the team is improving its improving its workflow when it comes to resolving issues.

Project SymPy is an open-source project in GitHub and is a Python library used for symbolic mathematics and is one of the projects that is being tracked by default by the Metrics Dashboard service.

Figure 6. Sympy: Line chart for issue density and KLOC against month

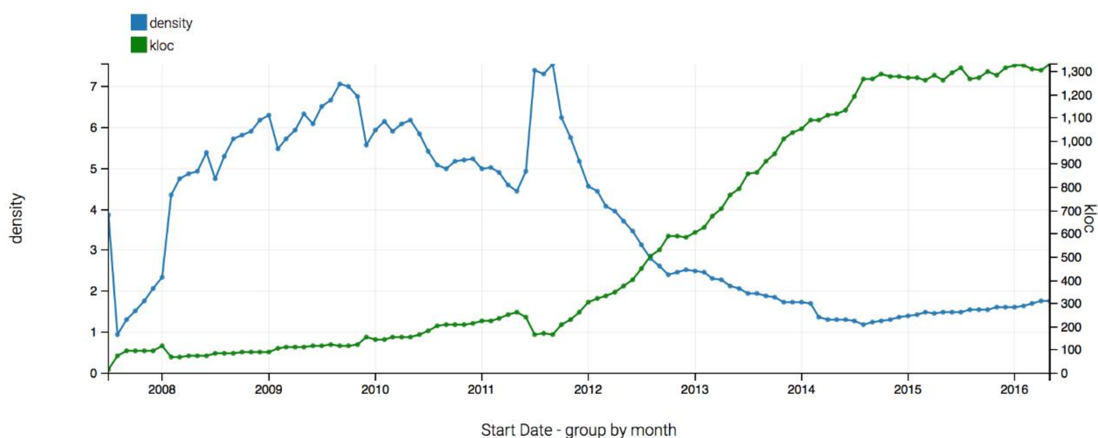


Figure 6, shows the issue density and KLOC for the project against month. At first glance, we notice that the KLOC or module size has increased significantly since 2012 but the issue density has reduced during the same period. Normally, it is expected that as the module size increases the number of issues for a project will also increase, giving higher values of issue density. However, this may not always be the case. As seen from figure 7, the yellow line, which indicates the closed issues cumulatively added since the beginning of the projects' lifetime, shows significant increase compared to the issues opened (shown in the color red) shows a steeper increase staring from the year 2012. Therefore, this would lead us to the conclusion that the issues are being closed at a faster rate than the rate at which they are opened, which in turn reduces the issue density during that period.

Figure 7. Sympy: Line chart for issues grouped by week

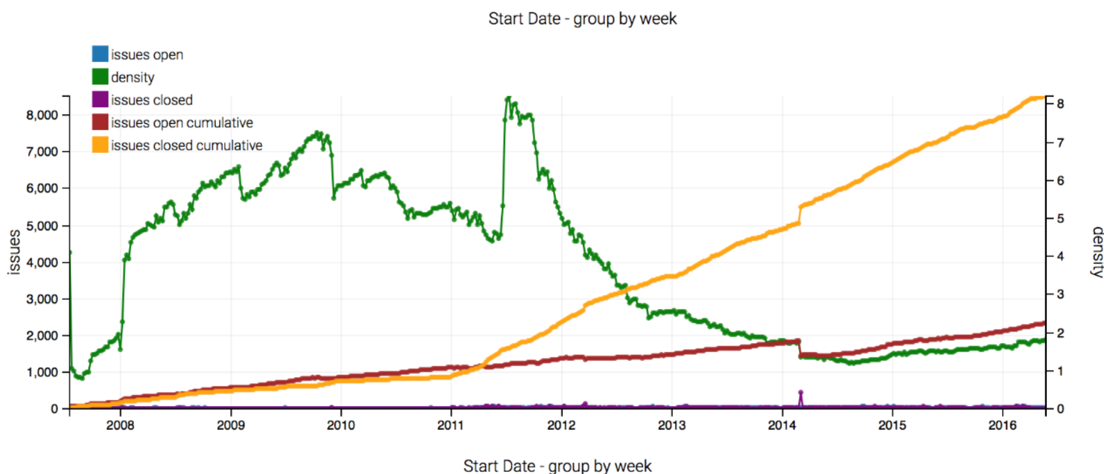
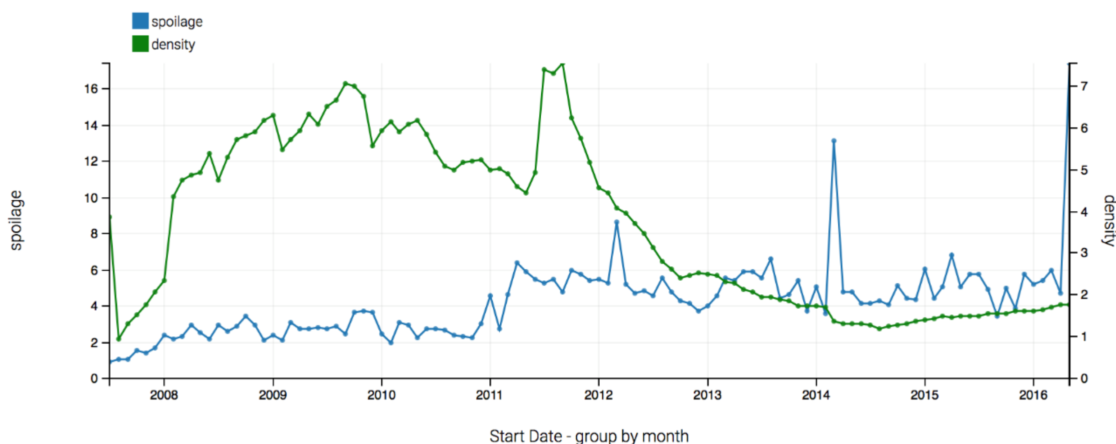


Figure 8, shows the spoilage for SymPy, one would expect that since the issue density has reduced and the larger number of issues are being closed than they are opened, the spoilage should also show significant reduction. It is interesting to note that this isn't always the case.

Figure 8. Sympy: Line chart for issue density and spoilage against month



Note that spoilage is a measure of how long it takes to fix an issue, therefore, even though a team manages to close issues at a fairly decent rate, if there are older issues in

the project that are still in the open state, this will significantly add to the spoilage and we may not see a drop in spoilage as seen for this project.

CHAPTER THREE

CONCLUSION

Simplistic measurements can cause more harm than good, but a combination of simplistic and derived metrics can serve as a useful tool at making software quality easily comprehensible to software developers. This thesis, aims at providing a clear idea of how the identified metrics are calculated and how we arrive at the results. A brief evaluation of the results obtained so far helps us identify areas in the time line where a project might have deviated from the norm. These results gives a team better insight on how the software development progresses over time. However, the metrics implemented by the Metrics Dashboard team, in no way, provides a thorough understanding of a projects' health, instead it serves as an initial step towards better understanding of a software development process which would help teams address many new challenges related to the development, deployment, and maintenance of reusable software.

Evaluation

The metrics implemented so far, have given us a basic idea of the development process for a project. The AWS server side implementation of the identified metrics can be used by teams with a simple request to the Metrics Dashboard team to track a project. The success of the work done so far depends heavily on whether the teams find the dashboard useful in identifying potential faults or areas that need to be worked on, for e.g. testing and logging issues, fixing older issues, reducing the time required to fix

issues. A sort of balance needs to be maintained between these metrics avoiding any high peaks and drops in the metrics. The metrics implemented so far by no means completely or fully understand a projects' health but when compared to simplistic measures like KLOC, count of issues, project contributors etc., these derived measures give a deeper view into a projects' development process overtime which could in turn help software development teams understand and improve software quality. The next steps to evaluation of metrics identification and usage is comprised of the following steps:

1. Evaluate whether CSE teams find the Metrics Dashboard useful: It is known that CSE software development teams embrace some aspects of Software Engineering. We can then capitalize on this to gauge interest in the idea of using metrics. It is key to understand what information an SE team is looking for while using the Metrics Dashboard service. Since the three metrics implemented so far depend on popular measures like KLOC, Issues, time to fix issues, this should serve as a useful addition to the already popular metrics.
2. Evaluate the effect of the Metrics Dashboard on software quality and software process: Software metrics serves as a useful tool in monitoring software development process, so it is key to track the effects these measures have on the maintenance of existing software modules or development of newer modules.
3. Add new metrics as they become necessary: Substantial interest in metrics is expected about reported defects (via the issue tracker in GitHub) over time and the mean time to resolve (fix) issues over time. While there are a large number of metrics that we could include in the dashboard, we will focus on metrics that can

be derived from information already collected by the tools projects are currently using.

4. We will migrate towards using Apache Spark, a cluster computing platform which serves as a general purpose engine for large scale data processing. The reason being that, GitHub allows a maximum of 5000 requests per hour (also called rate limit) for an authenticated request. Each request to GitHub API gathers information about a project and is useful in computing the derived metrics. This rate limit won't pose a problem for smaller projects, however, for larger CSE projects with a rate limit of 5000 the metrics computation and storage could take hours, which is not a feasible option. We plan to overcome this delay by cloning the repository locally and computing KLOC with the help of Apache Spark. We will still be using GitHub API to gather information on issues.

We aim not to tag projects as being good or bad, instead we want to ensure that teams focus on following good software engineering practices and we hope that our initial attempts at Metrics Dashboard will help achieve this goal.

LIST OF REFERENCES

[1] J. Carver and G. Thiruvathukal, “Software engineering need not be difficult,” tech. rep., Workshop on Sustainable Software for Science: Practice and Experiences, 2013.

[2] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Up- per Saddle River, NJ: Addison-Wesley Professional, 3 edition ed., Oct. 2012.

[4] G. Hohpe and B. Woolf, Enterprise *Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston: Addison-Wesley Professional, 1 edition ed., Oct. 2003.

[5] A. Gupta, O. P. N. Slyngstad, R. Conradi, P. Mohagheghi, H. Ronneberg, and E. Landre, “A Case Study of Defect-Density and Change-Density and their Progress over Time,” in *11th European Conference on Software Maintenance and Reengineering, 2007. CSMR ’07*, pp. 7–16, Mar. 2007.

[6] S. M. A. Shah, M. Morisio, and M. Torchiano, “An Overview of Software Defect Density: A Scoping Study,” in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1, pp. 406–415, Dec. 2012.

[7] S. Gala-Prez, G. Robles, J. M. Gonzlez-Barahona, and I. Herraiz, “Intensive metrics for the study of the evolution of open source projects: Case studies from Apache Software Foundation projects,” in *2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 159–168, May 2013.

[8] S. M. A. Shah, M. Morisio, and M. Torchiano, “Software defect density variants: A proposal,” in *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pp. 56–61, May 2013.

[9] N. Nagappan and T. Ball, “Static analysis tools as early indicators of pre- release defect density,” in *27th International Conference on Software Engineering, 2005. ICSE 2005*. Proceedings, pp. 580–586, May 2005.

[10] E. Bouwers, A. V. Deursen, and J. Visser, “Towards a Catalog Format for Software Metrics,” in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, WETSoM 2014, (New York, NY, USA), pp. 44-77 ACM, 2014.

[11] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*. 2013

VITA

Shilpika was born and raised in Mangalore, India. Shilpika has her Bachelor of Engineering degree in Electronics and Communication from Visvesvaraya Technological University, Belagavi, Karnataka State, India, in the year 2010. Following this, she worked for two years and 10 months, as a software engineer in Tata Consultancy Services (TCS), India. She moved to the United States in the year 2013 and the curiosity to explore more about Computer Science lead to her pursuing the degree of Masters in Computer Science at Loyola University Chicago. At Loyola, Shilpika worked as a Teaching Assistant, before giving that up for a Research Assistantship position which lead to this thesis. Her team won the second runner prize at the Internet and Televison Expo (INTX) hackathon. Currently, she works as an intern at Argonne National Laboratory.