Empirical analysis of the *bug triaging* and *time to review*: A case study of OpenStack.

Dorealda Dalipaj
Universidad Rey Juan Carlos
LibreSoft
SENECA Project
Madrid, Spain
dorealda.dalipaj@urjc.es

Gema Rodriguez-Perez
Universidad Rey Juan Carlos
LibreSoft
GSyC
Madrid, Spain
gerope@libresoft.es

Jesus M. Gonzalez-Barahona
Universidad Rey Juan Carlos
GSyC/Libresoft
Madrid, Spain
jgb@gsyc.es

Daniel Izquierdo Cortazar
Universidad Rey Juan Carlos
and Bitergia
GSyC/Libresoft
Madrid, Spain
dizquierdo@bitergia.com

ABSTRACT

Code review is an excellent source of metrics that can be used to improve the software development process. Metrics benefits varies from measuring the progress of a development team to investigating into software development policies and guidelines.

In this paper, we describe an empirical study of some of the *absolute metrics*, specifically *review process metrics*. Our case study is the large open source cloud computing project `OpenStack`. We bring evidence of *code reveiw response time* by quantifying the time spent by developers to identify the bug reports in the issue tracking system (*bug triaging*) and the time they spent to carry out the reviewing process (*time to review*) in the code review system. Last, we contrast our findings with data taken from traditional software inspection conducted on a Lucent project and from open source software code review on six projects, including AMD, Microsoft, and Google-led projects.

Our approach uses a reverse engineered model of the code review process, and extracts key information from the issue tracking and review systems. The metrics we bring evidence, do not involve subjective context but are material fact. As such, they can be recorded to trace the efficiency and effectiveness of the code review function.

Keywords

Review process metrics, software engineering, open source, review response time, quantitative evidence, OpenStack.

1. INTRODUCTION

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

The objective of code review is to detect development errors which may cause vulnerabilities, and hence give rise to an exploit. Code review is characterized as *"a systematic approach to examine a product in detail, using a predefined sequence of steps to determine if the product is fit for its intended use"* [1].

The formal review or inspection according to Fanagan's [2] approach required the conduction of an inspection meeting for actually finding defects. Different controlled experiments showed that there were found no significant differences in the outcome of the review process when comparing meeting-based with meetingless-based inspections [3, 4].

Another study [5] proved that the time for conducting the review process and it's outcome significantly improved with meetingless-based approaches.

As a result a wide range of mechanisms and techniques for code review were developed, which results in nowadays in the *modern code review* process. From the tool based static analysis [6, 7, 8] which examines the code in the absence of input data and without running the code, to the widely adopted modern code review [9, 10, 11] which aliging with the distributed nature of many projects is asynchronous and frequently supporting geographically distributed reviewers. Because of the many uses and benefits, code reviews are a standard part of the modern software engineering workflow.

It is generally accepted that the performance of code review process is affected by a variety of factors, some of which are external to the technical aspects of the code review itself [12]. Furthermore code review performance is associated with the effort spent to carry out the process.

2. PURPOSE OF THE STUDY

Code review is an excellent source of metrics [13] that can be used to improve the software development process. There are two distinct classes of these software metrics: relative and absolute. While relative metrics are a representation of an attribute that cannot be directly measured and are subjective and reliant on the context of where the metric was derived, absolute metrics are numerical values that describe a trait of the code that do not involve subjective context but

are material fact.

Thus, the purpose of this study is to bring empirical evidence by quantifying two absolute metrics of the code review process: the time spent for the bug triaging and the time to review the code.

Software code review is practiced on a diverse set of software projects that have drastically different settings, cultures, incentive systems, and time pressures. In an effort to characterize and understand these differences, previous studies [14, 15] have examined open source and non open source projects like Android OS, Chromium OS, Bing, Office, MS SQL, and projects internal to AMD. We contrast our findings on the time to review of our study with the data obtained from the study of these projects.

Our primary conjecture is that if some of the code review metrics in different projects have become similar as the projects have naturally evolved, then such characteristic may be indicative of convergent practices that represent generally successful and efficient methods of review. As such, these can be prescriptive to other projects choosing to add peer review to their development process.

Our overarching research question is *what is the code review process response time?* We operationalize this question for the two absolute metrics of code review process:

1. *What is the amount of time that developers need for bug triaging?*
2. *What is the actual amount of time that developers need to close a review?*

With the exception of the first question, the second parameter of review have been and is still being studied in many experiments [2, 16, 17, 18]. Our contribution is to compare a large diverse set of projects of an open source cloud computing project, such as OpenStack, on these parameters.

In the remainder of this paper, we first describe the necessary background notions for our work (section 3). Next, we describe the methodology (section 4), then present the results of the research questions (section 5). After comparing the results (section 6), we discuss threats to validity and future work (section 7 and section 8),we finish with acknowledgements (section 9).

3. BACKGROUND

This section provides background information about the bug tracking and code review environments of OpenStack and the tools for obtaining data from their repositories.

OpenStack is a free and open source set of software tools for building and managing cloud computing platforms. Because of its open nature, anyone can add additional components to OpenStack to help it to meet their needs. Actually, in OpenStack there are more than 200 active projects. The OpenStack community has identified 9 key components that create the *core* of OpenStack. These components are officially maintained by the OpenStack community: Nova, Swift, Cinder, Neutron, Horizon, Keystone, Glance, Ceilometer, and Heat. Therefore we will display the results dicided by the 9 core components of OpenStack and categorise the rest as Other Projects.

OpenStack uses Launchpad issue tracking system, a repository that enables users and developers to report defects and feature requests. It allows such a reported issue to be triaged and (if deemed important) assigned to team members, to discuss the issue with any interested team member and to track the history of all work on the issue. During these issue discussions, team members can ask questions, share their opinions and help other team members.

OpenStack uses Gerrit, a dedicated reviewing environment, to review patches and bug fixes. It supports lightweight processes for reviewing code changes, i.e., to decide whether a developer's change is safe to integrate into the official Version Control System (VCS). During this process, assigned reviewers make comments on a code change or ask questions that can lead to a discussion of the change and/or different revisions of the code change, before a final decision is made about the code change. If accepted, the most recent revision of the code change can enter the VCS, otherwise the change is abandoned and the developer will move on to something else.

To obtain the issue reports and code review data of these ecosystems, we used the data set provided by González-Barahona et al. [19]. They developed the MetricsGrimoire tool to mine the repositories of OpenStack, then store the corresponding data into a relational database. We make use of their issue report and code review data sets [20] to perform our study.

4. METHODOLOGY

To investigate our research questions we use a reverse engineered model of the code review process.

We first extracted code review data from the Launchpad[1] issue tracking system and the Gerrit[2] code review repository.

We then pre-processed the data, identified the factors to measure the metrics, and performed our analysis.

The extraction and processing of the data operations were both automatic. And in order to ensure the quality of data, after every heuristics applying, we manually analysed the selection picking up a number of random elements.

Finally, we compared our results with that of the previous study [14] and draw our results.

4.1 Data Extraction

In previous work, we described the extraction process and resulting data for Launchpad and Gerrit, the data is also available for other researchers [18]. This work did not involve analysis of the data. In the remainder of this section, we discuss what constitutes a review for the project and briefly describe how we extracted code review data for each metric.

4.1.1 Bug triaging time - Launchpad

This metrics measures the time that developers need to identify an issue, that has been reported, as a bug. Analysing the Launchpad work flow, we came across a pattern in the evolution of reports states, with regards to confirming new bugs:

> a) when a ticket, stating a possible bug, is opened in Launchpad, its status is set to *New*;
>
> b) if the problem described in the ticket is reproduced, the bug is confirmed as genuine and the ticket status changes from *New* to *Confirmed*;

[1]link: activity.openstack.org/dash/browser/data/db/tickets.mysql.7z

[2]link: activity.openstack.org/dash/browser/data/db/reviews.mysql.7z

c) only when a bug is confirmed, the status then changes from *Confirmed* to *In Progress* the moment when an issue is opened for review in Gerrit.

Thus, we analysed the Launchpad repository searching for tickets that match with this pattern. These are the tickets that have been classified as bug reports. Once identified, we extracted them in a new repository for further inspection.

Our results showed that, begining from 2011 up to March 2016, in Launchpad, 64.895 tickets out of 99.134 have been classified as bugs. Hence approximately 65.5% of the total tickets in Launchpad are have been reproduced as genuine bugs, and an issue for fixing them was opened in Gerrit.

At this point we are able to quantify the time that developers spend in identifying bug reports as the distance in time between the moment when the ticket is first inserted in Launchpad up to the moment it is *Confirmed* as a genuine bug.

You can see the numbers and percentages of the identified bug reports in OpenStack, divided by projects, in fig. 1.

The percentages of reported issues (tickets) classified as bugs in OpenStack, grouped by projects.

	Total Tickets	Bugs	Percentage of Bugs
Nova	13544	8251	**60.9%**
Swift	1809	1059	**58.5%**
Cinder	4582	3035	**66.2%**
Neutron	8134	5504	**67.7%**
Horizon	5713	3743	**65.5%**
Keystone	4313	2607	**60.4%**
Glance	3074	1932	**62.8%**
Ceilometer	1977	1352	**68.4%**
Heat	3423	2416	**70.6%**
Others	52546	34996	**66.6%**

Figure 1: The percentages of reported bugs in OS - From July, 2011 - March, 2016.

4.1.2 Review time - Gerrit

The next step is to link the tickets that we have already extracted from the issue tracking system with their respective review the Gerrit code review system. Traceability of this linkage is not a trivial task in OpenStack.

To detect the links between ticket and reviews, we referred to the information that is contained in the comments of the tickets. When a review receives a proposal for a fix, or a merge for a fix, it is reported in the comments of the respective issue. Precisely, a merge comment looks like the following:

Reviewed: https://review.openstack.org/100018
Committed: https://git.openstack.org/cgit/openstack/nova/commit/?id=be58dd8432a8d12484f5553d79a02e720e2c0435
Submitter: Jenkins
Branch: master ...

In the first line, we are provided with the identification of the issue in Gerrit.

The first problem that arises from the comments is that, for different tickets, they are a summary of some commit history. If this is the case, we will find more than a review to match with the issue we are looking for within the body of the comment, while the commit itself is not a merge in the master branch of the project that originated the ticket, consequently not the correct result.

However there is a fixed format of the comments that reports a merge (which is the one you see from the example above). In this format, the information related to the review is stated at the very beginning of the comment. Manually analysing the tickets in Launchpad, we have seen that they are found in the first 6 rows of the comment.

Thus the first step is trunking the comments, so that we extract exactly the 6 first lines from every one of them. This way, we will identify the right review.

At this point we are able to quantify the time to review in OpenStack as the distance in time from when the first patch is uploaded in Gerrit up to when a change is merged to the code base.

The table below (fig. 2) shows the number and percentage of tickets from Launchpad linked with the respective review in Gerrit.

Our approach was able to link approximately 81% of the tickets from Launchpad to their corresponding review.

Number of tickets merged	Number of tickets linked	Percentage of tickets linked
76238	61573	**80.8%**

Figure 2: The percentages of tickets linked with its counterparts review in OpenStack - From July, 2011 - March, 2016.

Analysing the missing merges from the resulting dataset by picking up arbitrarily tickets, we found that the reason they could not be linked is that some of the gerrit identification numbers, to wich tickets are related by a merge, are not found in the Gerrit database. For example:

the ticket https://review.openstack.org/#/q/topic:bug/1253497 is merged in the issue https://review.openstack.org/#/c/92547/, but the issue id 92547 is not found in the Gerrit database.

However, we would note that, thanks to the heuristics applied, there are no false positives in the resulting linkage. Thus the correctness of the dataset is satisfied.

5. RESULTS

In this section we expose the results that we have obtained for our research questions.

5.1 RQ1. What is the amount of time that developers need for bug triaging?

We computed the time for identifying the bug reports as discussed in *4.1.1.*

Afterwards, we calculated the median effect size across all Open Stack projects in order to globally rank the metrics from most extreme effect size, and last the quantiles.

We discoverd that the median time for identifying a bug report in Open Stack (Launchpad) is 1.97 hours.

Additionally, we expose the time to identify the bug reports for all the tickets divided per quartiles. Our results show that 25% of the issue reports are triaged in less than 4.62 minutes, 50% of the issue reports are triaged in less than 19.7 hours, and that 75% of the issue reports are triaged in approximately less than 3 days.

The results are shown in the table below (fig. 3):

Median time for bug triaging:

1.97 hours < 1 days

Quantiles:

0.25	277.0 sec	4.62 min
0.50	7105.0 sec	1.97 hours
0.75	260286.5 sec	3.01 days

Figure 3: The median time to classify a bug report across all projects in OpenStack - From July, 2011 - March, 2016.

5.2 RQ1. What is the actual amount of time that developers need to close a review?

We computed the time to carry out the review process as discussed in *4.1.2*.

We calculated the median across all OpenStack projects.

We discovered that the median time for reviewing is 52.2 hours (less than 2 days). We can additionally say that 25% of the reviews are completed in less than 8.21 hours, 75% of the reviews are completed in less than 213.8 hours, and 50% of the reviews are completed in from 8.21 hours up to 213.8 hours, with IQR (InterQuartile Range) equal to 205.54 hours (approximately 8.5 days).

Our results show that the time to review in OpenStack, along the various years of it's history, that goes from July 2011 up to March 2016, is under control. See results in the table below (fig. 4):

OpenStack	Time to review in Hours	Time to review in Days
Year 2011	3.6	0.2
Year 2012	16.8	0.7
Year 2013	36.6	1.5
Year 2014	78.2	3.3
Year 2015	57.9	2.4
All History	**52.2**	**2.2**

Figure 4: Time to review in OpenStack through its history - From July, 2011 - March, 2016.

(*Note*: given that the history of the year 2016 is relatively short, we decided to integrate it to 2015 in the results)

In the graph shown in figure 5 you can see the trends of the time to review for the different years of the OpenStack history spread through all the projects we are analysing (9 core projects, and the remaining in the Other Projects category).

From these results we can say that during 2011 and 2012 the time to review is under control. But during 2013, 2014, 2015 (includes history of 2016 from January to March) the time to review suffers some peaks with the highest value belonging to Nova, Glance, Heat, and Neutron.

In table exposed in figure 6 you can find the numeric values of the results shown in fig. 5. The time is measured in hours, reminding that the history of 2016 included in the *Year 2015* column. The values in bold denotes the peaks in the time to review behaviour for both project and year.

6. COMPARISON OF THE RESULTS AND CONCLUSIONS

As we previously mentioned, we are going to compare our results to the one obtained from other studies ([14, 15]) in an effort to characterize and understand the differences. These studies expose the results of the time to review for some of the AMD, Microsoft, and the Google-led projects, with the median calculated per month.

We bring our results to their metric of measurement (fig. 7) and obtained a time to review for Gerrit of 15.5 hours as the mean time to review scattered accross its projects.

	Nova	Swift	Cinder	Neutron	Horizon	KeyStone	Glance	Ceilometer	Heat
2011	1,67	0,00	0,00	1,029	0.0	0,00	0,34	0,00	0,00
2012	2,52	1,13	4,00	4,94	0,36	3,29	3,15	3,11	0,08
2013	26,05	7,54	10,40	6,29	19,18	20,97	26,32	13,55	8,44
2014	67,93	12,87	28,58	31,84	23,51	32,70	23,84	20,35	52,29
2015	59,32	15,23	25,12	22,20	13,57	20,96	34,05	16,20	20,03
Total(h)	**22,04**	**8,80**	**19,04**	**16,68**	**10,95**	**16,30**	**16,67**	**15,18**	**13,84**

Figure 7: Time to review in OpenStack projects.

Then we positioned the Gerrit time to review in the results obtained from the studies (fig. 8):

	Time to review (hours)
Bing	14.7
OpenStack	**15.5**
Google Chrome	15.7
AMD	17.5
Office	18.9
SQL	19.8
Android	20.8
Lucent	240

Figure 8: Comparison.

The conclusions we obtained from our analysis is that not only in OpenStack the time for bug triaging is short but also the review intervals are short rejecting the claim that *the median time from a review being requested to receiving*

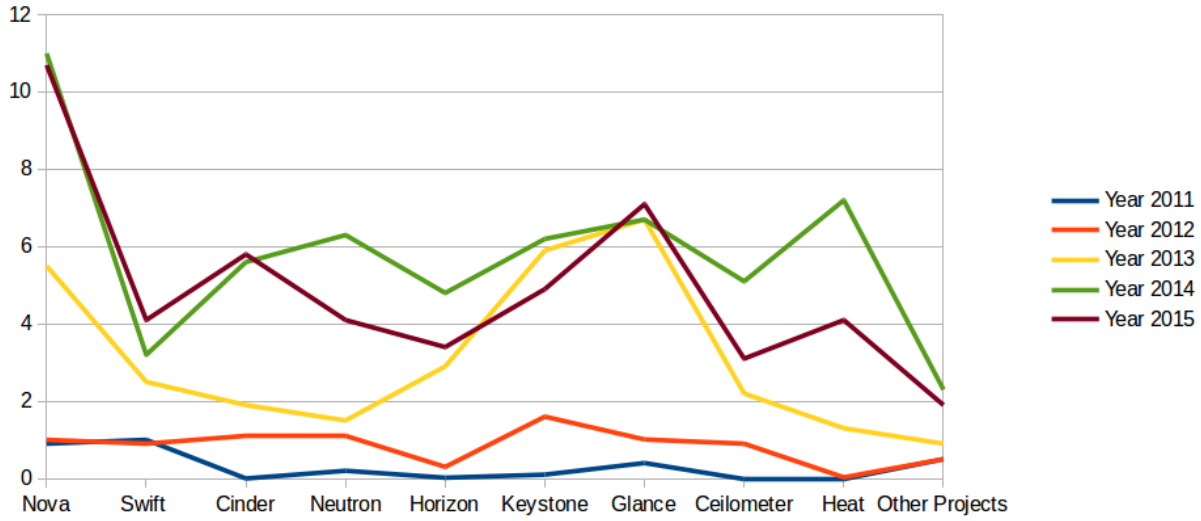

Figure 5: Time to review in OpenStack History scattered through the projects.

Project | Year	Year 2011	Year 2012	Year 2013	Year 2014	Year 2015	All History
Nova	21.7	24.4	**131.9**	**264.2**	265.7	125.1
Swift	24.3	23.2	60.2	77.8	99.5	67.7
Cinder	0	25.9	47.9	**135.2**	**138.6**	93.4
Neutron	3.9	26.5	35.9	**150.7**	95.5	86.3
Horizon	0.4	7.0	70.4	114.9	81.5	67.9
Keystone	1.9	38.3	**141.9**	**149.6**	119.7	102.9
Glance	8.4	24.2	**160.1**	**161.9**	**169.9**	101.9
Ceilometer	0	20.9	52.9	123.5	73.9	72.2
Heat	0	0.7	31.7	**171.9**	98.8	74.4
Other Projects	11.5	12.1	22.5	55.8	45.3	42.1
Open Stack	3.6	16.8	36.6	78.2	57.9	52.2

Figure 6: Time to review in OpenStack History scattered through the projects.

all necessary sign-offs is about 24 hours, with many lasting days if not weeks [21].

Furthermore the results found in fig. 8 show that despite differences among projects, the characteristics of the response time to review in the code review process have independently converged to similar values which we think indicate general principles of code review practice.

Additionally, based ont the results of fig. 8, we can state that contemporary code review is performed regularly and quickly.

7. THREATS TO VALIDITY

Due to the elaborate filtering that we performed in order to link two repositories (bug repository, and code review), the heuristics used to find the relations between them are not 100% accurate, however we used the state-of-the-practice linking algorithms at our disposal. Recent features in Gerrit show that clean traceability between version control and review repositories is now within reach of each project, hence the available data for future of this study will only grow in volume.

8. FUTURE WORK

There are several other metrics that characterize modern code review that we would like to investigate. One of them is what influences the time to review (fig. 5).

Analysing the time to review accross the projects of OpenStack in various years (fig. 9) of its history and the number of issues reviewed accross these same projects scattered through the years (10), we found that these two factors do not follow the same trends, indicating that other metrics influence the code review process.

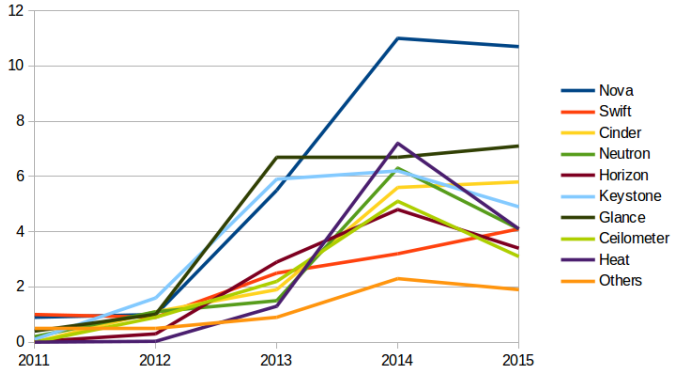

Figure 9:

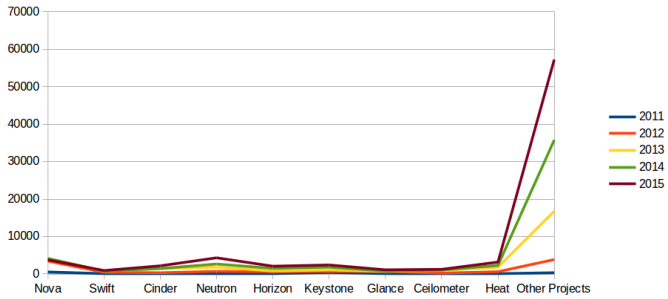

Figure 10:

9. ACKNOWLEDGMENTS

We would like to thank the SENECA EID project under Marie-Skodowska Curie Actions and Spanish Government, which are funding the authors working with this research. Additionally a special thanks goes to Bitergia for the Metrics Grimoire tool which realised the extraction of the dataset of our study.

10. REFERENCES

[1] D. L. Parnas and M. Lawford. Inspection's role in software quality assurance. In Software, IEEE, vol. 20, 2003.

[2] M. E. Fagan. Design and Code inspections to reduce errors in program development. In IBM Systems Journal 15 pp. 182-211, 1976.

[3] P. M. Johnson, and D. Tjahjono. Does Every Inspection Really Need a Meeting? In Empirical Software Engineering, vol. 3, no. 1, pp. 9-35, 1998.

[4] P. McCarthy, A. Porter, H. Siy et al. An experiment to assess cost-benefits of inspection meetings and their alternatives: a pilot study. In Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results, 1996.

[5] A. Porter, H. Siy, C. A. Toman et al. An experiment to assess the cost-benefits of code inspections in large scale software development. In SIGSOFT Softw. Eng. Notes, vol. 20, no. 4, pp. 92-103, 1995.

[6] W. R. Bush, J. D. Pincus, D. J. Sielaff. A static analyzer for finding dynamic programming errors, Softw. Pract. Exper. , vol. 30, no. 7, pp. 775-802, 2000.

[7] Hallem, D. Park, and D. Engler, Uprooting software defects at the source, Queue, vol. 1, no. 8, pp. 64-71, 2003.

[8] B. Chess and J. West, Secure Programming with Static Analysis, 1st ed. Addison-Wesley Professional, Jul. 2007.

[9] N. Kennedy. How google does web-based code reviews with mondrian. http://www.test.org/doe/, Dec. 2006.

[10] A. Tsotsis. Meet phabricator, the witty code review tool built inside facebook. http://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath- penned-these-words/, Aug. 2006.

[11] Gerrit code review - https://www.gerritcodereview.com/

[12] Baysal, O., Kononenko, O., Holmes, R., Godfrey, M. W. (2015). Investigating technical and non-technical

factors influencing modern code review. Empirical Software Engineering, 1-28.

[13] Code Review Metrics. Open Web Application Security Project. Open Web Application Security Project. Archived from the original on 2015-10-09.

[14] Peter C. Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE2013). ACM, New York, NY, USA, 202-212.

[15] Amiangshu Bosu and Jeffrey Carver. 2013. Impact of Peer Code Review on Peer Impression Formation: A Survey. Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2013. Baltimore, MD, USA, 133-142.

[16] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. ACM Transactions Software Engineering Methodology, 7(1):4 1-79, 1.

[17] P. C. Rigby, D. M. German, and M.A. Storey. Open source software peer review practices: A case study of the apache server. In ICSE: Proceedings of the 30th international conference on Software Engineering, pages 541-550, 2008.

[18] Dalipaj Dorealda. A Quantitative Analysis of Performance in the Key Parameter in Code Review - Individuation of Defects. International Conference on Open Source Systems (OSS) 2016, Doctoral Consortium.

[19] J. M. Gonzalez-Barahona, G. Robles, and D. Izquierdo-Cortazar. The metricsgrimoire database collection. In 12th Working Conference on Mining Software Repositories (MSR), pages 478-481, May 2015.

[20] activity.openstack.org/dash/browser/data/db/.

[21] Czerwonka Jacek, Michaela Greiler and Jack Tilford. Code Reviews Do Not Find Bugs. How the Current Code Review Best Practice Slows Us Down. Proceedings of the 2015 International Conference on Software Engineering. IEEE Publisher, 2015.