# How long does it take to fix the code: A case study of OpenStack.

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

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

## 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 analyse 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 review process 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 the results of similar analysis from traditional software inspection conducted on the Lucent project and from open source software code review on six projects, including AMD, Microsoft, and Google-led projects.

This analysis is one of the intermediate phases of a current research line. Conducted and funded under the SENECA, a EU project, the scope of the research is to analyse the process quality characterising key performance indicators of the software development processes.

## Keywords

Review process metrics, software engineering, open source, OpenStack.

## 1. INTRODUCTION

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, finding and fixing defects, of the review process when comparing meeting-based with meetingless-based inspections [3, 4]. Additionally the time to complete the process improved significantly with meetingless-based inspections [5].

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 [6, 7, 8, 9, 10, 11]. 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

The purpose in this paper is to bring evidence by quantifying two absolute metrics from the code review process: time needed for bug triaging and time needed to review the code. The two metrics involve the most important phases of the code fixing process. It starts when an issue reporting a possible problem is started, until the moment when a fixing for the bug is merged to the code base. These metrics describe attributes of the process that do not involve subjective context but are material facts.

The other phases composing the code fixing process are related to management, thus are relative metrics. Non the less, they too have their influence in the improvement of the code fixing process, but matter of factly they are difficult to record and manage objectively.

But as previously mentioned, this is only an intermediate analysis of a broader study in progress. The ultimate scope of the work in progress, is to analyse the whole process from intention to code review of the software development and deployment. By identifying the most inlfluencing phases of this process, the aim is to dentify the factors that influence the outcome and come up with a collection of best practices for industry.

The analysis and management of these specific metrics have lots of benefits for the industry. Software companies can measure the progress of a development team in their practice of application development. They can identify sections where the development practice is weak or sections where it is strong. It further gives a software company the

*WOODSTOCK '97 El Paso, Texas USA*
© 2017 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

ability to address the root cause of the weaknesses or extract the cause of the strength by featuring practices within a developed solution.

This may give rise to reformulation of the software development policies and guidelines redefining practices for a more efficient and effective process. Furthermore, metrics related to the performance of the developers community in the accuracy of the review process and the code review function performance in terms of efficiency and effectiveness, can also be recorded.

Over the years, code review has been applied on a diverse set of software projects that have totally different settings, incentive systems and time pressures. In an effort to characterize and understand these differences, previous studies [13, 14] 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 the code from our study, with the data obtained from the study of these other projects, noticing a very interesting idiosyncrasy.

Furthermore, the metric of time to review have been and is still being studied in many experiments [2, 15, 16]. Our contribution is to analyse and compare on this parameter a large diverse set of projects of an open source cloud computing project such as OpenStack.

## 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.

OpenStack uses Launchpad issue tracking system, a repository that enables users and developers to report defects and feature requests.

OpenStack uses Gerrit, a dedicated reviewing environment, to review patches and bug fixes. It supports lightweight processes for reviewing code changes. If a code change is accepted it can be integrated into the official Version Control System (VCS), otherwise the change is abandoned.

To obtain the issue reports and code review data of these ecosystems we used MetricsGrimoire [17], a toolset for mining the repositories of OpenStack.

## 4. METHODOLOGY

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

We first extracted code review data from Launchpad issue tracking system and Gerrit code review system repositories.

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 [13] and draw our results.

### 4.1 Data Extraction

In this section we briefly describe how we extracted the data for each metric.

#### *4.1.1 Bug triaging time - Launchpad*

This metrics measures the time from the moment a ticket stating a possible defect is reported, up to the moment when the defect reported is confirmed as a real bug. We extracted the data analysing the evolution of the state of a ticket, with regards to confirming new bugs:

a) when a ticket, stating a possible bug, is opened in Launchpad, its status is set to *New*;

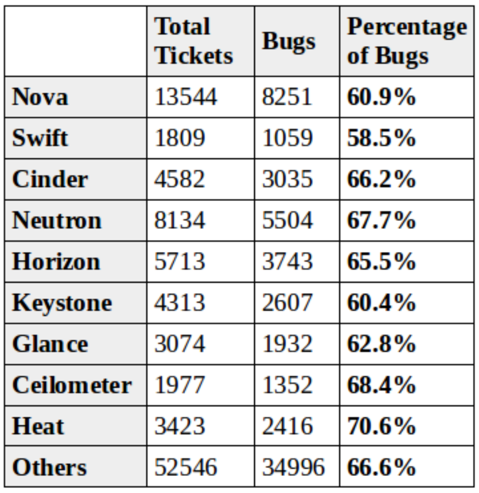
b) if the problem described in the ticket is reproduced, the bug is confirmed as genuine and the ticket status changes from *New* to *Confirmed*;

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.

Once identified the tickets that match with the previous pattern, we extracted them in a new repository for further inspection.

The results showed that, from 2011 up to March 2017, in Launchpad, 64.895 distinct 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.

You can see the numbers and percentages of the identified bug reports in OpenStack, grouped by the 9 core projects, with the rest of the components reppresented by the category 'Others', in Figure 1.



| | **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, 2017.**

#### *4.1.2 Review time - Gerrit*

This metric measures the time from when the first patch of an issue is uploaded in Gerrit up to when the code fixing is merged into the code base.

In order for us to achieve this, we need to link the tickets that we already extracted form Launchpad with their re-

| OpenStack | Time to Review (Days) |
|---|---|
| **Year 2011** | 0.2 |
| **Year 2012** | 0.7 |
| **Year 2013** | 1.5 |
| **Year 2014** | 3.3 |
| **Year 2015** | 2.4 |
| **Year 2016-2017** | 2.2 |

**Figure 3: Time to review over the years across OpenStack - From July, 2011 - March, 2017.**

spective review in Gerrit. Traceability of this linkage is not a direct task in between Launchpad and Gerrit.

To detect the links between ticket and reviews, we referred to the information that is contained in the comments of the tickets, precisely in merge comments, where we are provided with the identification of the issue that has fixed the bug in Gerrit.

A further caution at this point, to identify the right review, was to select the commits that have merged the fix in the master branch of the project that originated the ticket.

Therefore, we preprocessed the comments of the tickets to extract the information needed to link.

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 the time to run the bug triaging and the time to review the code.

We calculated the median effect size across the Open Stack projects in order to globally rank the metrics from most extreme effect size, and lastly the quantiles.

We discoverd that the median time to triage a bug report in Open Stack (Launchpad) is 0.9 days, while the median time to review the code (Gerrit) is 2.2 days.

The results of the quartiles are shown in the table below (fig. 2):

| Tickets (%) | Bug Triaging Time (Days) | Code Review Time (Days) |
|---|---|---|
| 25 | 0.01 | 0.7 |
| 50 | **0.9** | **2.2** |
| 75 | 27.8 | 11.6 |

**Figure 2: The median time to triage a bug and time to review in OpenStack - From July, 2011 - March, 2017.**

Additionally to have a better prospect of what happens over the years and in the various projects of Openstack, we calculated the results of time to review metric over the years across OpenStack (fig. 3), and over the years across different projects (fig. 4). As the history for year 2017 is relatively, we decided to integrate it with the year 2016, aggregating the result.

The results are measures in *days*.

In the table displayed in Figure 4, you can see the trend of

| Project | 2011 | 2012 | 2013 | 2014 | 2015 | 2016-2017 |
|---|---|---|---|---|---|---|
| Nova | 0.9 | 1.0 | **5.5** | **11.0** | 10.7 | 5.2 |
| Swift | 1.0 | 0.9 | 2.5 | 3.2 | 4.1 | 2.8 |
| Cinder | 0 | 1.1 | 1.9 | **5.6** | **5.8** | 3.9 |
| Neutron | 0.2 | 1.1 | 1.5 | **6.3** | 4.1 | 3.6 |
| Horizon | 0.02 | 0.3 | 2.9 | 4.8 | 3.4 | 2,8 |
| Keystone | 0.1 | 1.6 | **5.9** | **6.2** | 4.9 | 4.3 |
| Glance | 0.4 | 1.01 | **6.7** | **6.7** | **7.1** | 4.2 |
| Ceilometer | 0 | 0.9 | 2.2 | 5.1 | 3.1 | 3.0 |
| Heat | 0 | 0.03 | 1.3 | **7.2** | 4.1 | 3.1 |
| Other Projects | 0.5 | 0.5 | 0.9 | **2.3** | 1.9 | 1.8 |

**Figure 4: Time to review (measured in days) over the years divided by projects in OpenStack - From July, 2011 - March, 2017.**

the time to review during the history of OpenStack, across all it's projects (9 core projects, and the remaining represented in the Other Projects category).

From these results we can say that during the years 2011, 2012 and 2016-2017 the time to review seems under control. But during 2013, 2014 and 2015, time to review suffers some peaks with the highest value belonging to Nova, Glance, Heat, and Neutron.

## 6. COMPARISON OF THE RESULTS AND CONCLUSIONS

As we previously mentioned, we are going to compare our results, on time to review, with the one obtained from other studies ([13, 14]) in an effort to characterize and understand the differences. These previous studies have analysed time to review for some of the AMD, Microsoft, the Google-led projects, which practice the modern meeting-less code review method, and Lucent, which practices the traditional meeting based reivew system. Their results calculated the median values for this metric.

| Project | Time to Review (Days) |
|---|---|
| AMD | 0.7 |
| Android | 0.9 |
| Bing | 0.6 |
| Google Chrome | 0.7 |
| Office | 0.8 |
| OpenStack | 2.2 |
| SQL Server | 0.8 |
| Lucent | 10 |

**Figure 5: Comparison of the time to review values for various projects.**

Looking at the values in fig. 5, we see that except for Lucent, where meeting-based code review is performed, the other values are comparable to each other. Our assumption is that if some of the code review metrics, in this case time to review, in different projects are comparable as the projects have progressed independently, then this aspect may be indicative of practices that represent a generally successful and efficient pattern of review.

Additionally, based ont the results of fig. 5, we can state that contemporary code review is performed regularly and quickly.

There are other characteristics of the projects we compared that we have taken into consideration when we assume that their time to review values are comparable. One the most important is the fact that OpenStack is backed up and maintained by a globally distributed community of developers and then the number of reviews they practice.

The only measure we have, at the best of our knowledge, for *bug triaging time* is an indication provided by [18], an experiment conducted with the support data from Microsoft. They found that *bug triage takes up a large amount of developers resources and it goes from about 24 hours to many lasting days, if not weeks.* Our contribution here is to bring some numbers on how long this process takes in a large cloud computing project, such as OpenStack.

The final purpose of this paper is to highlight, from the code review process, the two metrics of *bug triaging time* and *time to review*, and recognize them as key metrics to supervise during the review process. You can't control what you can't measure. And by controlling these metrics, not only the effectiveness of code review can be quantified and the progress of a development team can be measured, but also policies and guidelines, which can improve the development process, may arise. As such, these metrics can be prescriptive to other projects taking into consideration to add code review to their development process.
Additionally, in a continuous integration and deployment environment, such as OpenStack, the role that the two activities of bug triaging and code review have on the overall process of code review becomes of vital importance. Not only the quality of the code under development must be assured, but also the time waiting to take possesion of the new code must be relatively as short as possible.

## 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

Taking into consideration all the above discussion, our work in progress and future work is to continue analysing more phases from the software development process.

There are several metrics that characterize the code review process that we are currently investigating, and how this metrics can improve the process itself.

The purpose of our work is raising specific metrics as key performance indicators (KPIs). These KPIs when controled, are able to improve the software development process either by discovering software development policies or guidelines, and by setting configurations of a development team.

## 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] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.