Does a Code Review Tool Evolve as the Developer Intended?

Osamu Mizuno and Junwei Liang

Abstract In this study, we intend to assess the improvements of Gerrit. The central concern is "Does Rietveld evolve into Gerrit as the developers intended?" To answer this question, we first compare qualitative features of two code review tools. We then conducted an interview with a developer of Gerrit and obtained the developer's original intention of improvements in Gerrit. By analyzing mined data from code review logs, we try to explain the effects of improvements quantitatively. The result of analysis showed us that the improvements of Gerrit that the developer is expected are not observed explicitly.

Keywords Software evolution · Code review · Open source development · Empirical study

1 Introduction

Mining software repositories has become a current trend of software engineering. Much research using mining technique has been done to discover various issues in software engineering [1]. Repositories mining research often focuses on the quality of software. For example, detection of fault-prone software modules is one of good field of repositories mining [2, 3]. In fact, the number of fault-prone module detection approaches using mining technique has rapidly increases in last 5 years.

In order to assure the quality of software, early detection of defects is recommended. The code review is one of effective ways for such early detection of defects in software [4]. The code review activities include various useful insights for software quality. However, especially in open source software (OSS) developments, records of code review merely remained in a systematic way. Logs of reviews were

O. Mizuno (✉) · J. Liang
Software Engineering Laboratory, Graduate School of Science and Technology, Kyoto Institute of Technology, Kyoto, Japan
e-mail: o-mizuno@kit.ac.jp

J. Liang
e-mail: j-liang@se.is.kit.ac.jp

mostly on the mailing-list, and researchers needed so many efforts to obtain data from mailing-lists or unstructured data [5].

Recently, for the effectiveness of the code review process in OSS development, tools for code review have been developed. One common weakness of mail based review was the lack of linking between patches and the version control system [6]. The review with tools provides this linkage. For example, a diff made against an older version of a file can be updated by selecting the most reviewed version within a tool, or an approved review can be immediately committed into the version control system.

Not so many open source projects, however, adopt review with tools. Chromium is one of them, which is a successful open source browser project used the code review tool called Rietveld. Rietveld is an open source code review tool based on Mondrian, the internal code review tool used by Google. Both of Rietveld and Mondrian was created by Guido van Rossum who is best known as the author of the Python programming language.

Rietveld was a good tool for code review. The developers, however, appended patches to Rietveld to improve features they want. As a result, they built a new tool for code review from a scratch. The documents of Gerrit says the background as follows [7]:

> Gerrit Code Review started as a simple set of patches to Rietveld, and was originally built to service AOSP. This quickly turned into a fork as we added access control features that Guido van Rossum did not want to see complicating the Rietveld code base. As the functionality and code were starting to become drastically different, a different name was needed. Gerrit calls back to the original namesake of Rietveld, Gerrit Rietveld, a Dutch architect.

For this reason, Gerrit is a successor of Rietveld, and thus it is natural that Gerrit has improved features from Rietveld. Like Rietveld and Gerrit, software is sometimes improved and evolved into another new software. Such improvement is done by various reasons, for example, user's requirements, developer's intuition, performance issues, and so on. Although the effect of improvement should be evaluated, it is hard to measure and thus remained not evaluated.

In this study, we aim to evaluate the improvements of Gerrit. The basic question is "Does Rietveld evolve into Gerrit as the developers intended?" To answer this question, we first compare qualitative features of two code review tools. We then conducted an interview with a developer of Gerrit and obtained the developer's original intention of improvements in Gerrit. By analyzing mined data from code review logs, we try to clarify the effects of improvements quantitatively.

The rest of this paper is organized as follows: Sect. 2 describes the code review process with tools and comparison of code review tools. Section 3 shows the result of interview with a developer of Gerrit and research questions. Mined data from code review repositories are explained in Sect. 4. Section 5 investigates the research questions. The threats to validity are discussed in Sect. 6. Finally, Sect. 7 concludes this study.

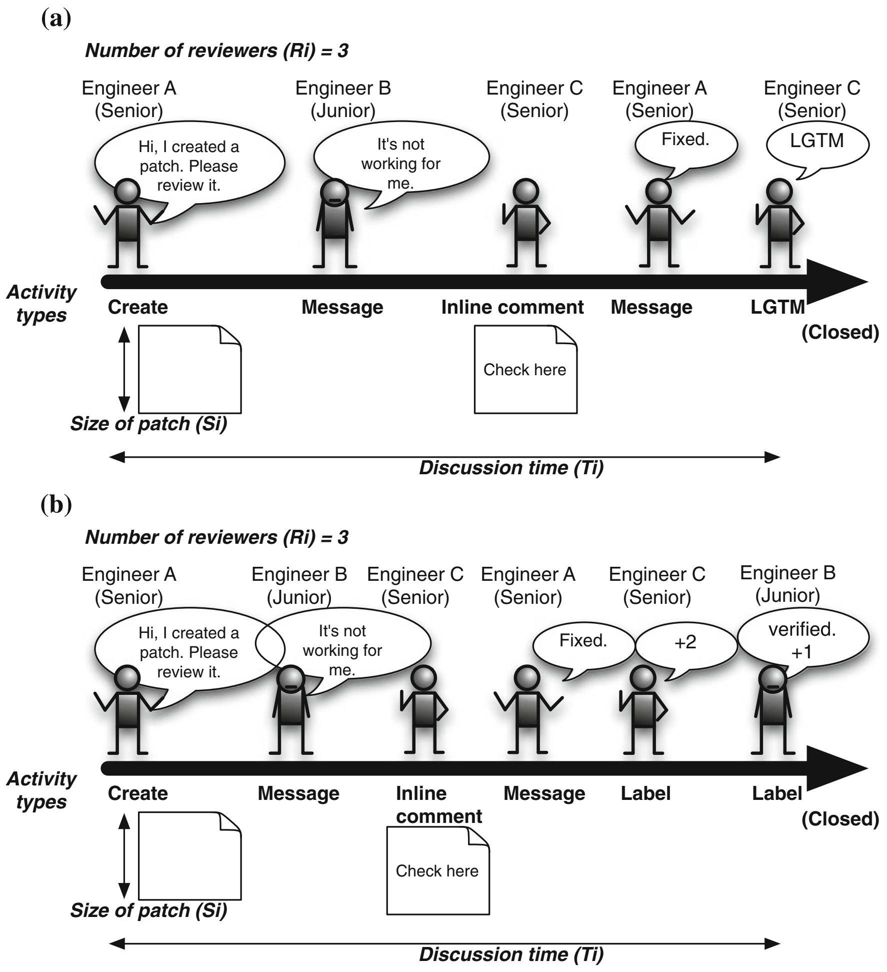

Fig. 1 An example of code review process. **a** Review process with Rietveld. **b** Review process with Gerrit

2 Code Review Tools

2.1 Code Review Process

Figure 1 shows an example process of code review using code review tools. Figure 1a shows a case of Rietveld and Fig. 1b shows a case of Gerrit. In both tools, the fundamental process is similar. First, a developer create a review issue with his/her patch to the source code. Then some developers (reviewers) look at the patch and write messages or inline comments to the creator if they find any problems.

Table 1 Features of Rietveld and Gerrit

Feature	Rietveld	Gerrit
Web-based	Yes	Yes
Side by side diff	Yes	Yes
Inline comments	Yes	Yes
VCS	SVN	Git
Implementation	Python on Google App Engine	Java on J2EE and SQL DB
Access control	No	Yes
Keyboard shortcuts	Yes	Yes
Code review labels	No. "LGTM" in a message	Yes $+2, +1, -1, -2$ labels
Code review messages	Yes	Yes
Verify labels	No	Yes $+1$ and -1
Search options	Simple	Powerful
Command line tools	Yes	Yes (more elegant)
Show commit dependencies	No	Yes
Multiple projects	No	Yes
Issue submit guideline	No	Yes based on labels

The creator reviews the comments, make changes if necessary, and publishes updated patch in response to the comments. Once all comments have been addressed either through code or discussion, the reviewers will approve the review issue and close the review issue after merge the patch into the central code repository.

The differences between Rietveld and Gerrit are seen in "LGTM" and "Label" in Fig. 1a and b. The guidelines of operating these tools also differ each other. The qualitative difference between them is shown in Sect. 2.2.

2.2 *Qualitative Comparison*

Table 1 shows features of Rietveld and Gerrit. Since Gerrit is a successor of Rietveld, most features in Gerrit are enriched from Rietveld.

By comparing features between Rietveld and Gerrit shown in Table 1, the improvement in Gerrit is summarized the following 4 features: (1) Access control, (2) Code review labels, (3) Verify labels/messages, (4) Search options.

The main motivation of Gerrit development is integration with Git. As a result of Git integration, the access control feature is attained to Gerrit.

Shawn Pearce, the main developer of Gerrit, said:

> Access controls in Gerrit are group based. Every user account is a member of one or more groups, and access and privileges are granted to those groups.

Rietveld has no explicit label feature, but has "Quick LGTM (looks good to me)" feature. By pressing the "Quick LGTM" button, the reviewer can quickly post a message with "LGTM". By extending this future, a label system is implemented in Gerrit.

The labels attached on an issue are essential information to determine to accept or decline the code review. Such labels are called as "code review labels" in these code review tools. Rietveld. however, does not have the label, but has the message named "LGTM (Looks Good To Me)". Reviewers attach this message if they find that the code is good for commit. In the development of Chromium, there is not a clear criterion to determine to accept or decline an issue, i.e. if most of participant attached "LGTM", the issue may be accepted.

On the other hand, Gerrit implemented the label system with labels of "+2 (Looks good to me, approved)", "+1 (Looks good to me, but someone else must approve)", "0 (No score)", "−1 (I would prefer that you didn't submit this)", and "−2 (Do not submit)". The +1 and −1 levels are just an opinion where as the +2 and −2 levels are approving or abandoning the review issue. Only review issues having at least one +2 and no −2 labels can be approved. There is no meaning to accumulate these label value. Two +1 labels do not equate to one +2 label [8].

Since Gerrit is a successor of Rietveld, the main purpose of Gerrit is the improvement of code review processes in Google OSS projects. Such improvements are, however, not evaluated quantitatively.

3 Research Questions

3.1 Interview with the Developer

For clarifying the original developer's intentions to develop Gerrit as a successor of Rietveld, we interviewed with Shawn Pearce, who is the main contributor of the development of Gerrit. The questions and answers are as follows:

Q1: What is the motivation to adopt access control of developers in Gerrit? Is it for reducing spammy or trivial commits?

A2: AOSP determined it wanted senior engineers who were conducting code reviews to be able to tick an "approve" box on the web, but not be distracted by the mechanics of downloading a patch, applying it to a local tree, and pushing that to the central server.

To allow anyone to submit a change without doing the manual "download + patch + commit + push" steps from the command line we introduced access controls to determine who can tick the "approve" box, and who can push the "submit" button to deliver the code to the central server automatically.

It enabled a productive boost for the senior engineers that were mostly conducting the code reviews.

Q2: What was the main purpose to adopt code review labels (+2, +1, ...)? We guess that it is for a criterion of patch commits (i.e. if a patch have +2 or larger label, the patch can be committed.) What is the purpose of introducing the verify labels and messages?

A2: Yes. It was a means to realize the access controls to tick the "approve" box. Once we decided we needed an approve box, we built the code-review label.
We then realized we wanted maybe a junior engineer we trust to download the patch, compile it, verify unit tests still pass, etc. and have them mark a different box that says "yup, code compiles!".
This become the verified box. So the code-review OK and the verified OK needed to be done by different people (code-review: senior engineer, verified: junior engineer), so these were different labels.
We then realized it looked bad in the community that nobody else could come along and say "yes I also like this patch". So we expanded code-review to be a range of -2 through 2. Trusted +senior engineers familiar with the project were given access to use the -2 and $+2$ end of the range. Everyone else (even a random user that stumbles on the site) can use the $-1 \cdots + 1$ range to say "yes me too I also like this".

3.2 Research Questions

From the interview with Shawn Pearce, we aim to clarify whether the productive boost for senior engineers is achieved or not. To do so, we state the following research question:

RQ Do code review activities become more productive in Gerrit-based project?

RQ aims to investigate whether the entire review activity in Gerrit-based project is more productive or not.

4 Data Retrieval

4.1 Target Projects

In order to compare the review processes between Rietveld and Gerrit, we need to find review repositories that adopt Rietveld or Gerrit. For Rietveld, we used the Chromium project,[1] a development of Chromium browser by Google and Google Web Toolkit

[1] http://www.chromium.org.

Table 2 Format of issue data from Rietveld

Item	Type	Description
i	Nominal	Unique ID for the issue
R_i	Counting	The number of reviewers who appear in the issue i
T_i	Counting	The discussion time for the issue i
S_i	Counting	The size of patch code is the issue i (lines)
N_i^{LGTM}	Counting	The number of "LGTM" messages in code review.

Table 3 Format of issue data from Gerrit

Item	Type	Description
i	Nominal	Unique ID for the issue
R_i	Counting	The number of reviewers who appear in the issue i
T_i	Counting	The discussion time for the issue i
S_i	Counting	The size of patch code is the issue i (lines)
L_i^{max}	Counting	The max value of labels in the issue i.

Table 4 Format of developer commitment data

Item	Type	Description
d	Nominal	Unique ID for the developer
i	Nominal	Unique ID for the issue
A_i^d	Nominal	The action that the developer d takes in the issue i

(GWT) project.[2] For Gerrit, we used the Android project,[3] a development of Android OS projects, and Qt project.[4]

4.2 Data Obtained

We obtained the log of code review activities by our mining tool for code review repositories [9]. We obtained various kind of data from the code review repositories. Tables 2 and 3 show the schema of data related to review issues. Table 4 shows the schema of data related to the developer's activities in review. These tables include only necessary information for later discussions.

[2] https://developers.google.com/web-toolkit/.

[3] http://source.android.com.

[4] http://qt-project.org.

5 Do Code Review Activities Become More Productive in Gerrit-based Projects?

5.1 Preliminary Questions

For investigating RQ, we define three preliminary questions as follows:

- How many engineers in review are there in Rietveld-based and Gerrit-based projects?
- How many review issues are there in Rietveld-based and Gerrit-based projects?
- What kind of activities did engineers do in review in Rietveld-based and Gerrit-based projects?

The first challenge is that, sometimes, a developer would like to have more than one accounts to commit to the project, which we called *developer aliases* issue. If we want to answer questions about developer, such as how many engineers involved in project, we should resolve the aliases issue first. We implemented a similar algorithm based on the *levenshtein edit distance* [10] with the approach proposed by Bird et al. [11] to automatically extract developer aliases. We also manually review the result from the web interface of our mining tool, and remove the alias links we thought were questionable.

Table 5 is an overview of data that we obtained for each project. It provide the answers of the first and the second questions. In more detail, the number of reviewers for each issue, R_i, is summarized in Table 6 by projects. The median of reviewers for each issue is one developer for both Rietveld-based projects and Gerrit-based projects. Interestingly, the same result has been also observed in other OSS projects using commit-then-review (CTR) patch review process [6]. Figure 2 shows distribution of R_i by projects.

From Table 5, we can see that the number of issues per unique reviewers varies by projects, not by code review tools. We cannot find significant difference on the number of review issues and the number of unique developers in this table.

Table 5 Overview of data obtained for each project

Project	Chromium (Rietveld)	GWT (Rietveld)	Android (Gerrit)	Qt (Gerrit)
Review issues	82,303	3,294	9,413	34,891
Number of unique reviewers	1,850	295	823	692
Number of issues per unique reviewer	44.49	11.16	11.52	50.42
Number of commitments	444,218	21,719	49,029	359,031
Duration	Sept. 2, 2008–May 24, 2011	Dec. 3, 2008–Aug. 18, 2012	Oct. 21, 2008–Oct. 26, 2012	May 17, 2011–Nov. 28, 2012

Table 6 Descriptive statistics for R_i for each project

	R_i (count)			
Project	Minimum	Median	Mean	Maximum
Chromium	0	1	1.270	19
GWT	0	1	1.249	7
Android	0	1	1.197	18
Qt	0	1	1.407	12

From Table 6 and Fig. 2, we can see the distribution of the number of reviewers for each issue, R_i. By comparing the Rietveld-based and Gerrit-based projects, there are more issues with $R_i = 1$ and less issues with $R_i = 0$ in Rietveld than in Gerrit. By investigating further for R_i, we found that there are many $R_i = 0$ issues in the abandoned issues of Gerrit-based projects, but there is not so many issues with $R_i = 0$ in Rietveld even in abandoned issues. It can be explained that many issues are immediately abandoned after creation in Gerrit.

Figure 3 shows that the ratio of action types for each project. "Create" shows an event that an engineer creates a review issue. "LGTM/label" shows that an engineer gives an "LGTM" message for an issue in Rietveld or a label for an issue in Gerrit. "Message" shows that an engineer gives a message for an issue. Finally, "Inline comment" shows that an engineer gives an inline comment for an issue.

We can see that the ratio of "LGTM/Label" is greater in Gerrit-based projects than Rietveld-based projects, while the "inline comment" activity is smaller in Gerrit-based projects than in Rietveld-based projects. This indicates that developers in Gerrit-based projects prefer to contribute to reviews by voting review or verify labels rather than add inline comments to patch code.

5.2 *Main Question*

For this research question, we measure how a project is productive by the discussion time for review issues. Generally speaking, if we need less time for an issue to process, we can say it is more productive. We thus translate original research question into as follows:

- Does an issue take less discussion time to review in Gerrit-based projects?

In order to investigate this research question, we define the following metrics:

- T_i: the discussion time for an issue i (h)
- S_i: the size of patch submitted for an issue i (LOC)

In order to consider the difficulty of an issue, we investigate the discussion time normalized by the changed lines of code for an issue, which is expressed as follows:

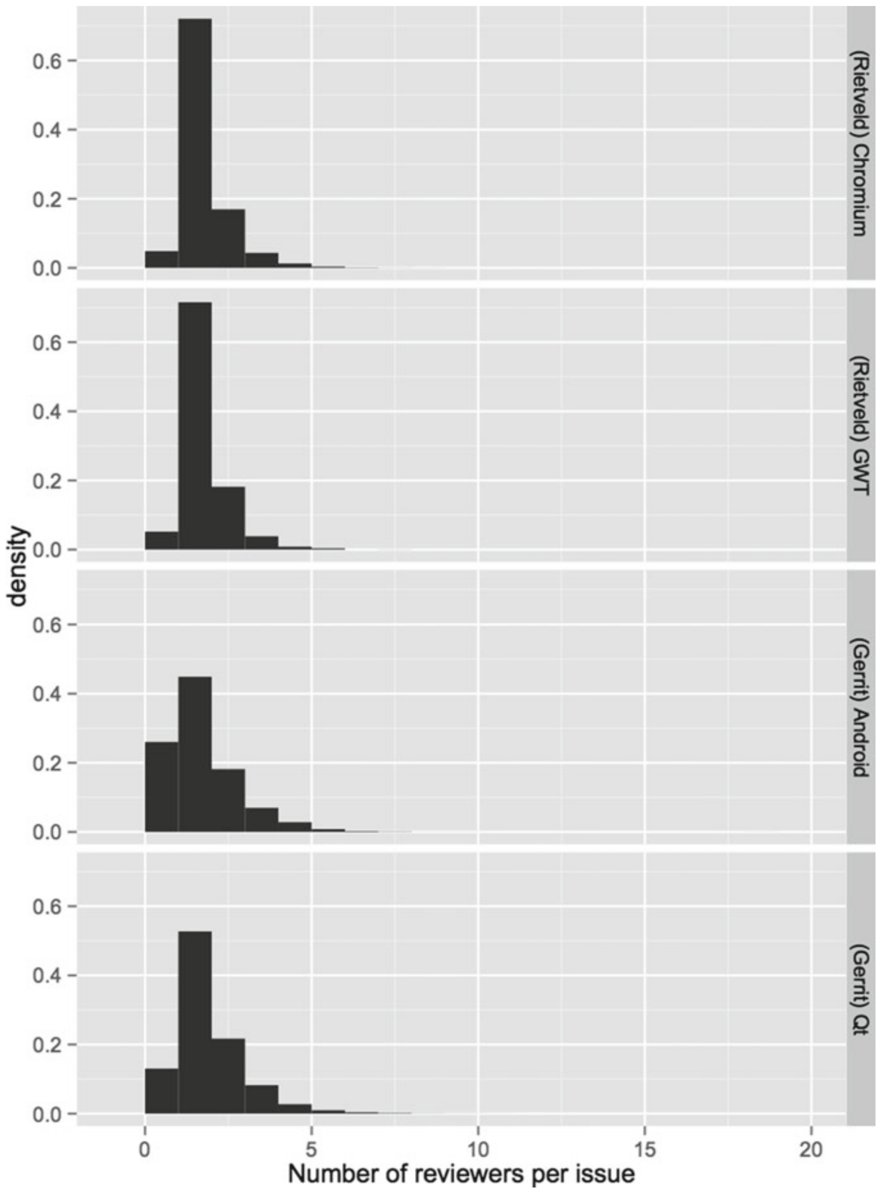

Fig. 2 Density histogram of R_is for all projects

$$\frac{T_i}{S_i}$$

Table 7 shows the descriptive statistics for T_i for each project. The median of T_i in Chromium is 2.199 h and GWT is 15.64 h, while it is 23.04 h in Android and 20.98 h in Qt. It should be noted that the median of T_i in Android is close to 24 h.

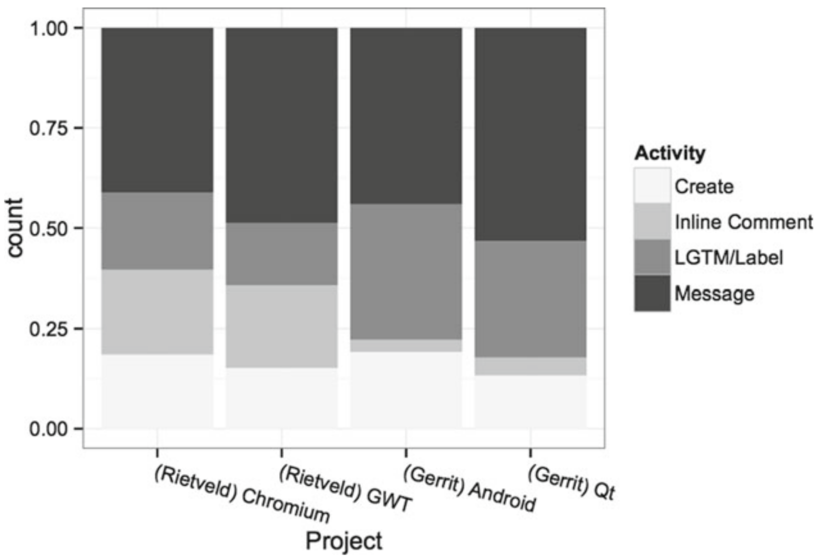

Fig. 3 Histogram of action types for each project

Table 7 Descriptive statistics for T_i for each project

	T_i (h)			
Project	Minimum	Median	Mean	Maximum
Chromium	0.009	2.199	53.73	8746
GWT	0.009	15.640	193.40	8240
Android	0.009	23.040	689.50	8751
Qt	0.009	20.980	136.40	8524

Table 8 Descriptive statistics for T_i/S_i for each project

	T_i/S_i (h/line)			
Project	Minimum	Median	Mean	Maximum
Chromium	0	0.098	4.067	5,211
GWT	0	0.134	12.510	2,325
Android	0	0.874	121.500	8,670
Qt	0	0.934	15.030	5,013

The descriptive statistics for T_i/S_i is shown in Table 8. Table 8 shows that median of T_i/S_i of Rietveld-based projects are significantly smaller than that of Gerrit-based projects.

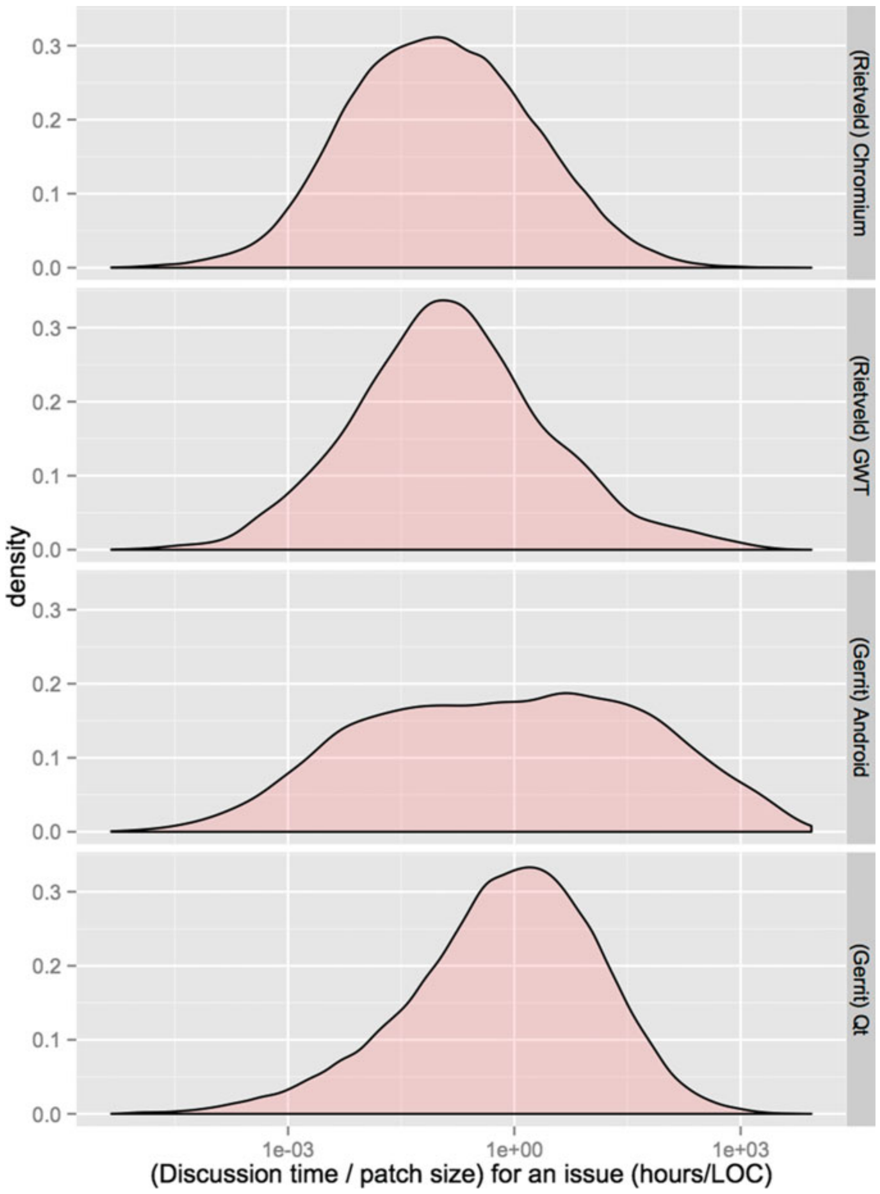

Fig. 4 Density histogram of T_i/S_is for all projects

Figure 4 shows a density histogram for T_i/S_i in all projects. Note that x-axis is shown in log scale. We can see that Gerrit-based projects require more discussion time to review a line of code than Rietveld-based projects.

In Gerrit-based project, it is recommended to pend an issue for 24 h to ensure any interested parties around the world have had a chance to commit. The document of Gerrit says as follows [7]:

Pending changes are likely to need at least 24 h of time on the Gerrit site anyway in order to ensure any interested parties around the world have had a chance to comment.

On the other hand, Rietveld has no guideline for reviewing time. This could result the discussion time longer in Gerrit-based projects than in Rietveld-based ones.

To conclude, we can say that the answer of this research question is "no". The discussion time for review issues in Gerrit-based projects is longer than that of in Rietveld-based projects.

5.3 Discussion

As Shawn Pearce told us in his interview, there is a notable improvement in Gerrit, the label system. We investigate the effect of the label system from the viewpoint of discussion time. To do so, we state the following question:

- Do LGTM messages in Rietveld and labels in Gerrit affect to discussion time?

Figure 5 shows the distribution of discussion time by the number of LGTM messages, N_i^{LGTM}, for Rietveld-based projects. Figure 6 shows the distribution of discussion time by the max value of labels, L_i^{max}, for Gerrit-based projects. Since LGTMs

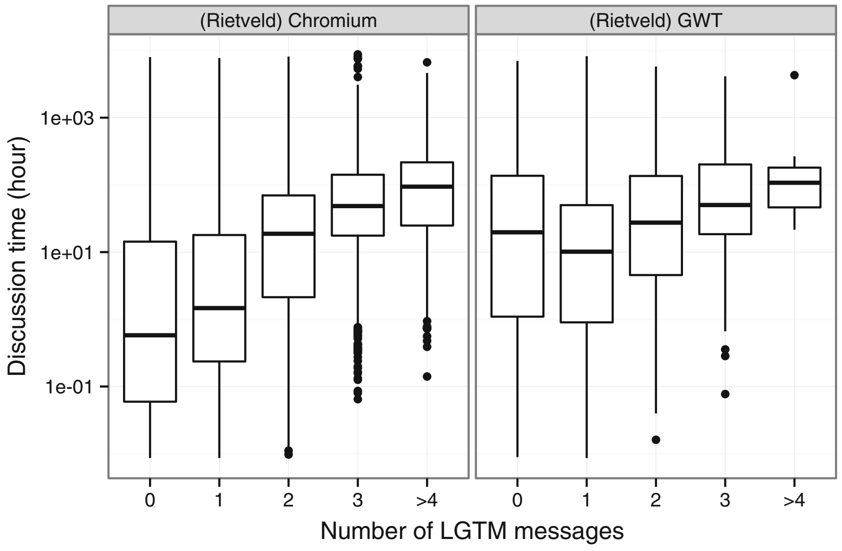

Fig. 5 Discussion time versus the number of LGTM messages for Rietveld-based projects

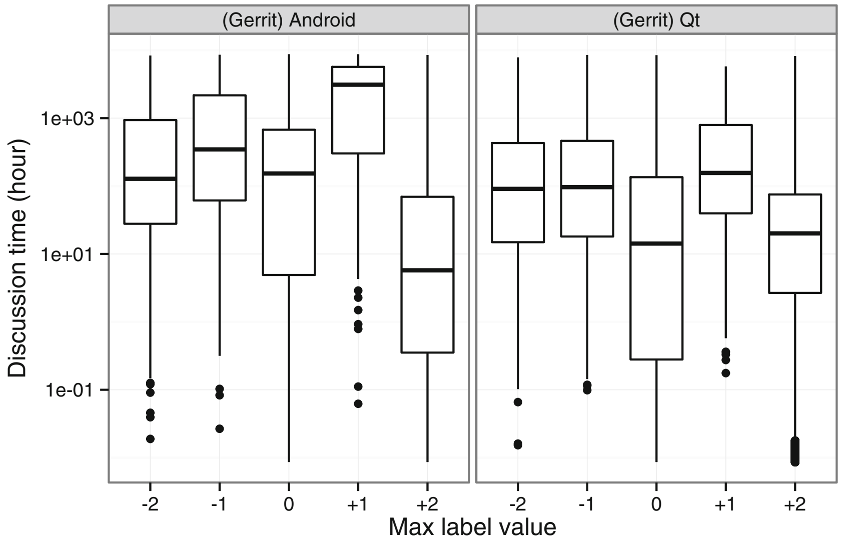

Fig. 6 Discussion time versus the number of labels for Gerrit-based projects

and labels are similar concept, we compare these four projects using LGTM and labels.

In Fig. 5, the discussion time increases according to the increase of LGTM messages. This indicates that if an issue got more LGTM messages, the discussion time becomes longer because there are no clear guideline to finish discussion based on the number of LGTM messages.

In Fig. 6, the discussion time decreases if the max label is +2 or 0. We should note that it is what the developer of Gerrit intended that the discussion time decreases in the case of +2. As we mentioned in Sect. 2.2, there is a guideline that an issue can be closed when it get at least one +2 label. We can say that the label +2 can reduce the discussion time for obviously acceptable issues. On the other hand, we need to investigate the case of $L_i^{max} = 0$ in which the discussion time decreases, too. For the label "0", the document of Gerrit says as follows [12]:

0 No score: Didn't try to perform the code review task, or glanced over it but don't have an informed opinion yet.

The label "0" means that no labels were added by any developers. We then looked at the status of reviews with $L_i^{max} = 0$ and found that 1880 of 1950 such reviews in Android were abandoned and 67 % were abandoned by the creator, 2141 of 2190 such reviews in Qt were abandoned and 76 % were abandoned by the creator. These findings show that developers in Gerrit-based projects often abandoned reviews by themselves even before anyone review that, which made the discussion time very short because no discussion occurred. The similar discussion can be applied to issues with $N_i^{LGTM} = 0$ in Rietveld.

We can conclude that the label system in Gerrit contributes to decrease the discussion time for issues with less problems, while the LGTM messages in Rietveld does not contribute to decrease the discussion time for issues.

6 Threats to Validity

We have several threats to validity in this study. This section discusses on such known threats.

In order to reduce individual variation as far as possible, we tried to find more projects using Rietveld or Gerrit. Unfortunately, we found that there are not so many optional projects for us when we consider project size.

The definition of senior engineers is mainly based on the previous study [13]. There are various definitions of senior engineers in both quantitative and qualitative ways. Since the definition affects the result of analysis directly, we need to refine the definition rigidly.

7 Conclusion

Our findings and contributions are summarized as follows:

- Developers in Gerrit-based projects prefer to contribute to reviews by adding review or verify labels rather than to add inline comments to patch code.
- The discussion time for review issues in Gerrit-based projects is longer than that of in Rietveld-based projects.
- The label system in Gerrit contribute to decrease the discussion time for issues by providing a clear guideline to when an issue can be closed, while the LGTM label in Rietveld does not function to decrease the discussion time for issues.

Acknowledgments The authors would like to express great thanks to Mr. Shawn Pearce, who willingly gave us the answer to our question related to Gerrit. The authors would like to thank Prof. Ahmed E. Hassan and members in Software Analysis and Intelligent Laboratory. This work was supported by JSPS KAKENHI Grant Number 24500038.

References

1. Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software change. IEEE Trans. Softw. Eng. **31**(6), 429–445 (2005)
2. Catal, C., Diri, B.: Review: a systematic review of software fault prediction studies. Expert Syst. Appl. **36**(4), 7346–7354 (2009). doi:http://dx.doi.org/10.1016/j.eswa.2008.10.027
3. Hata, H.: Fault-prone module prediction using version histories. Ph.D. thesis, Osaka University (2012)

4. Rigby, P.C., Storey, M.A.: Understanding broadcast based peer review on open source software projects. In: Proceedings of 33rd International Conference on Software Engineering, pp. 74–83 (2011)
5. Thomas, S.W.: Mining unstructured software repositories using ir models. Ph.D. thesis, Queen's University (2012)
6. Rigby, P.C.: Understanding open source software peer review: review processes, parameters and statistical models, and underlying behaviours and mechanisms. Ph.D. thesis, BASc. Software Engineering, University of Ottawa (2004)
7. Gerrit code review—system design. URL http://gerrit-documentation.googlecode.com/svn/Documentation/2.5.1/dev-design.html
8. Gerrit code review—a quick introduction. URL http://gerrit-documentation.googlecode.com/svn/Documentation/2.5.1/intro-quick.html
9. Liang, J., Mizuno, O.: Analyzing involvements of reviewers through mining a code review repository. In: Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, pp. 126–132 (2011). doi:http://doi.ieeecomputersociety.org/10.1109/IWSM-MENSURA.2011.33
10. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. (CSUR) **33**(1), 31–88 (2001)
11. Bird, C., Gourley, A., Devanbu, P., Gertz, M., Swaminathan, A.: Mining email social networks. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, pp. 137–143. ACM (2006)
12. Gerrit code review—access control. URL http://gerrit-documentation.googlecode.com/svn/Documentation/2.5.1/access-control.html
13. Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W.: The secret life of patches: a firefox case study. In: Proceedings of 19th Working Conference on Reverse Engineering, pp. 447–455 (2012)