# Code Review Analytics: WebKit as Case Study

Jesús M. González-Barahona, Daniel Izquierdo-Cortázar, Gregorio Robles, Mario Gallegos

**To cite this version:**

Jesús M. González-Barahona, Daniel Izquierdo-Cortázar, Gregorio Robles, Mario Gallegos. Code Review Analytics: WebKit as Case Study. 10th IFIP International Conference on Open Source Systems (OSS), May 2014, San José, Costa Rica. pp.1-10, 10.1007/978-3-642-55128-4_1 . hal-01373050

**HAL Id: hal-01373050**

**https://inria.hal.science/hal-01373050**

Submitted on 28 Sep 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

# Code review analytics: WebKit as Case Study

Jesus M. González-Barahona[1], Daniel Izquierdo-Cortázar[2], Gregorio Robles[1], and Mario Gallegos[3]

[1] GSyC/LibreSoft, Universidad Rey Juan Carlos {jgb,grex}@gsyc.urjc.es
[2] Bitergia dizquierdo@bitergia.com
[3] Universidad Centroamericana José Simón Cañas mgallegos@uca.edu.sv

**Abstract.** During the last years, most of the large free / open source software projects have included code review as an usual, or even mandatory practice for changes to their code. In many cases it is implemented as a process in which a developer proposing some change needs to ask for a review by another developer before it can enter the code base. Code reviews, therefore, become a critical process for the project, which could cause delays in contributions being accepted, and risk to become a bottleneck if not enough reviewers are available. In this paper we present a methodology designed to analyze the code review process, to determine its main characteristics and parameters, and to detect potential problems with it. We also present how we have applied this methodology to the WebKit project, learning about the main characteristics of how code review works in their case.

## 1 Introduction, motivation and goals

Code review is gaining importance in free, open source software (FLOSS) projects, as it started to gain relevance several years ago in proprietary software firms [2, 1]. Currently, most large FLOSS projects are using it in one way or another. Understanding how it is working, how it can be characterized with traceable, measurable parameters, and understating how it may affect to the relationships between actors in the project is becoming of great importance [4]. In this paper, we present a methodology that addresses these needs[1]. It starts by identifying traces from the review process in software development repositories such as source code management or issue tracking systems, and goes all the way to the characterization of performance and extension properties of the process. In particular, the following research questions are addressed: (Q1) To which extent can the review process based on traces in development repositories be characterized? (Q2) How can the evolution over time of the code review process be characterized?

The answer to Q1 is important because if those traces can be found, and automatically extracted from software development repositories, an automated

[1] Reproduction information and data sources of the study, according to [3], are available at http://gsyc.es/~jgb/repro/2014-oss-webkit-review

or semiautomatic methodology could be designed, implemented and deployed to track the evolution of the main parameters of the code review process. This would be a first step to build an automated dashboard that allows to better understand the process and for the continuous follow-up of those aspects by any interested party [5]. Q2 is focused on identifying parameters as simple as possible to calculate, but that capture information about important aspects of the evolution of the code review process. Again, if this can be done, instead of using a large collection of complex parameters, a small number automatically computed could be used. In our case, we have checked these two questions in the well-known WebKit project.

The next section presents the WebKit code review process, and provides a qualitative answer to Q1. Then, the methodology for the data retrieval and postprocessing is described in Section 3, including a quantitative answer to Q1. The analysis itself, with the answer to Q2, follows in Section 4. Section 5 is devoted to discussion and the analysis of the main threats to validity. The paper concludes with a section presenting the conclusions.

## 2 The code review process, and its traces

In WebKit, most significant source code contributions must go through a review process. However, activities considered trivial, or very basic maintenance issues (such as minor fixes due peculiarities of one of the platforms) can be committed directly.

### 2.1 Code review: is it possible to follow it in detail?

Anyone may send a contribution to WebKit. But usually the contribution must go through a review process, and be accepted by a reviewer. Both committers and reviewers are selected by previous WebKit committers and reviewers by a meritocratic, peer-approval process[2].

The contribution process[3] is centered around the Subversion repository and the Bugzilla system. Developers start by choosing or opening a new ticket in Bugzilla. The ticket may correspond to a bug report, a feature request, or something else. While working on it, developers compose a patch in their local working copy of the Subversion repository, including entries in changelog files, describing the changes and identifying the ticket. Then, it is submitted to Bugzilla with another script, where it is attached to its ticket.

Usually, upon submission to Bugzilla, code review is requested. This can be done by flagging the attachment to the ticket (as "Review?"), but it can also be requested by other means, such as in the project IRC channel. Unfortunately, only the flagging in Bugzilla leaves traces. Fortunately, flagging the ticket is

[2] `http://www.webkit.org/coding/commit-review-policy.html`
[3] `http://www.webkit.org/coding/contributing.html`

the most popular requesting method. The review request is not directed to a reviewer in particular, although the developer may try to get the attention of some of them by CCing them in the ticket update, or by directly addressing them somehow.

Reviewers deal with review requests according to their preferences. Once they have reviewed a contribution, they can decide to accept it, to ask for changes, or to reject it. Acceptance and rejection is signaled with a new flag ("Review+" or "Review-") to the ticket. When developers are asked for changes, they have to send a new patch for review with a new "Review?" flag. Changes may follow several iterations, which can in many cases be tracked by examining the review flags.

Once a contribution is accepted, it can be committed to the Subversion repository by any committer, or marked for automatic commit by the commit-queue bot. Therefore, only those developers who are also committers usually commit their own contributions. This means that the "committer" field in the Subversion commit record does not contain information about the real author or reviewer (and Subversion keeps no information about authorship).

### 2.2 Traces

Summarizing, the traces left by the code review process are:

- Changelog files in the Subversion repository. They include information for every commit, and at least author (usually including name and email address), Bugzilla identifier of the related ticket, and reviewer (if the commit was accepted after a code review process).
- Commit records in the Subversion repository. The committer information is not reliable, but useful information can still be extracted from the committed changelog files.
- Attachments and flags in the Bugzilla repository. Each contribution is usually submitted as an attachment to a ticket, and reviews are requested and granted usually setting flags in it. Detailed timing of all these operations is available, and some information about the person performing it. So, at least this information is available: time of review request ("Review+?") and results of the review process: acceptance ("Review+") or rejection ("Review-"), and Bugzilla identifier of those changing the state of the ticket.

Not always all of this information is available. However, commits with missing information is mainly from old reviews; since about 2005 a very large fraction of them have all data (see details in section 3). Using this information, a characterizations of the review process is possible:

- The most reliable information about authors and reviewers (i.e., name and email address) comes from the changelog files, since they are well documented.
- Alternatively, authors and reviewers could also be determined from their identities in Bugzilla ticketsA manual examination of a random collection shows

that both sources offer the same information, as usually scripts automatically include identities in both.

- Timing information is obtained from Bugzilla. The review process starts when a developer flags an attachment to a ticket as "Review?". The end of the process occurs when a "Review+" flag is set. If there are several requests, the timing of each "Review?" flag can also be determined.
- Authors and reviewers are linked to tickets thanks to the ticket identifier found in the changelog files.

With all this information the duration of reviews, and the number of iterations (number of review requests) can be tracked with great accuracy. Therefore, the answer to Q1 is, in principle, positive: the review process can be characterized with great detail at least in the above described terms. Some other aspects of the review process could be characterized with these data, such as the size of reviewed code (which could be extracted both from the commit record and the attachments to the ticket), the changes to code due to the review process (comparison of attachments to the ticket), etc.

## 3 Methodology

The methodology that we have used to characterize the code review process in WebKit is based on the following steps, similar to those described in [3]: data retrieval from development repositories into databases; clean-up, organization and sampling; and analysis. The first two steps are presented in this section.

### 3.1 Data sources and data retrieval

The study has been performed using data from the Bugzilla (issue tracking) system[4], and from the Subversion (source code management) repository[5] of the WebKit project. In the case of the Subversion repository, it has been accessed through a git front-end[6] which allows for complete access to all the information.

The git front-end to the Subversion repository was cloned on January 17th 2013 and includes information since August 24th 2001. Metainformation about all commit records (a total of 125,863) was obtained using CVSAnalY, from the MetricsGrimoire toolset[7]. This metainformation was used mainly for cross-validation, but is not really a data source for this study. The git clone was also used to extract information from changelog files. These files, which are spread through the source code tree, were identified, and their relevant information extracted, using an ad-hoc script based in part in the webkitpy library[8],

[4] `https://bugs.webkit.org`
[5] `http://www.webkit.org/building/checkout.html`
[6] `http://trac.webkit.org/wiki/UsingGitWithWebKit`
[7] http://metricsgrimoire.github.com
[8] `https://trac.webkit.org/browser/trunk/Tools/Scripts/webkitpy`

maintained by the WebKit project itself. This script retrieves the complete list of commits from the git clone, identifying and parsing for each of them the modified changelog files. From these files, it extracts the relevant fields: author, reviewer and Bugzilla ticket. A total of 117,079 entries in changelog files were identified this way.

The exact strategy followed by the script to determine changelog entries starts by obtaining a list of all commit identifiers (hashes) directly from the git front-end. For each of them, changelog files added or modified are identified, and their diff information obtained. All entries in those diffs are considered to be related to the commit. The author, reviewer and Bugzilla ticket identifier are retrieved and stored.

Information for all changes to all Bugzilla tickets was retrieved on January 29th 2013 using Bicho, from the MetricsGrimoire toolset. A total of 100,221 issues, and 976,879 ticket state changes were retrieved, with the first ticket dating from June 1st 2005 (there is a single older ticket from 2000, which seems to be an error, and was not considered).

### 3.2 Cleaning and organizing the data

A quick observation of the retrieved dataset shows how, as expected, committer information in the Subversion repository is unreliable. 24,406 commit records were found to have a committer which is not the author, according to the changelog information. Many of those are submitted by bots, who perform automatic commits of already reviewed code, or of small maintenance changes. For example, during 2012 about 22% of commits (7,079 out of 31,923) were performed by bots.

The analysis of the changelog files shows that they covered a very large fraction of all commits: only 8,784, or about 7% of the commits, are missing in the changelogs. The difference can be attributed in some cases to errors, but usually to minor maintenance commits, such as versioning commits, which are not considered to deserve an entry in changelog files. Although other approaches are possible, we considered only a single author for each commit. This meant that of the commits identified in changelog files, an additional 578 (about 0.5%) were ignored because they included information about more than one author (this usually happens when several developers collaborated in the code change).

The number of commits that included “Reviewed by” and similar entries was 74,290. On the other hand, the number of commits with changelog files that reference a Bugzilla ticket is 68,460. Both conditions (being reviewed and referencing a ticket) are fulfilled by only 60,991. Many of the commits that do not comply with both conditions correspond to the period before June 2005, when tickets were not introduced in the Bugzilla database. In some cases, a ticket is referenced in more than one commit: 55,649 unique tickets were found in changelogs. When looking for those tickets, some corresponded to non-public tickets (such as those used in some cases by Apple developers), being 54,501 the total number of tickets that we could use in our study.

The last step in selecting tickets is considering only those with complete information about the review process: we need to know when it was initiated, and when it finished. For that, we selected tickets flagged at least once as "Review?" (review request) and "Review+" (review approved).

Our study will consider tickets with review request after 2006 and before 2013. This will avoid the early days of the project when few information about code review is available in the Bugzilla system, and the final part of the sample, which would distort the final part of the evolution studies. For that period, we have a total of 75,179 commits marked as reviewed in the changelog files. Out of them, 61,867 (82%) reference a Bugzilla ticket, with 56,483 different tickets referenced. Of those, 55,303 (98%) are publicly available. Of those, 53,212 include at least one review request and approval: this is our final sample, which will be used for the following analysis.

## 4 Analysis over time

In order to characterize the evolution of the review process over time, after informal discussion with some WebKit developers, we have used the following parameters as they capture the most relevant aspects of how the code review process is changing:

- Bulk parameters: number of commits subject to code review, number of authors and reviewers involved, per month. They capture the "size" of the review process: how many actors are involved, how many actions ("review processes") they perform.
- Performance parameters: number of iterations (review requests needed) per review, delay from review request to reviewed. They capture how much effort is put into the review process (measured by iterations), and how much delay the process is causing (by measuring time-to-review).

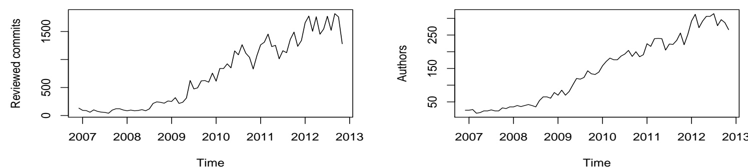


**Fig. 1.** Bulk parameters (authoring). Total number of tickets corresponding to review processes, and of active authors (review requesters) per month. Time of review request is used to determine the month for each ticket.

Figure 1 shows the bulk parameters for authoring. The number of reviewed commits is increasing clearly over time (from less than 400 per month before

mid-2009 to around 1,500 per month during 2012). The number of active authors per month is also increasing, following closely (although not always) the number of reviewed commits. In this case, the growth started a bit earlier than for commits, and shows what at first sight seems to be a linear trend.

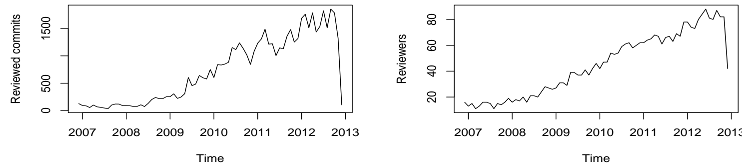


**Fig. 2.** Bulk parameters (reviewing). Total number of tickets corresponding to review processes, and of active reviewers (review approvers) per month. Time of review approved is used to determine the month for each ticket.

Bulk parameters for reviewing, shown in Figure 2, show very similar patterns. With a median delay of 151 minutes, times for asking for review and granting it are very close, which explains the almost equal shapes for commits. The growth in reviewers, on the contrary, is a bit slower than in the case of authors. While from 2008 to 2012 authors increased from around 50 to 250-300, reviewers grew only from about 20 to 80.

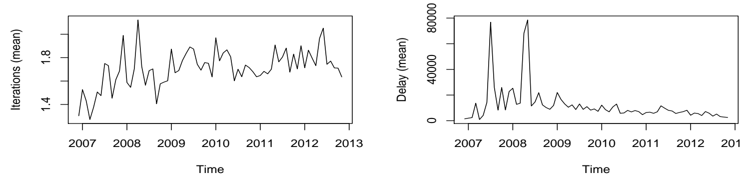


**Fig. 3.** Performance parameters (mean per month). Number of iterations (review requests for the same ticket) and delay (time from review request to review approval, in minutes).

Performance parameters tell about how the process is actually working over time. Figure 3 shows the evolution of the means: mean iterations and mean delay over time. In the case of iterations (number of cycles implying review requests) per ticket, although there is a lot of variance over time, a slowly growing trend seems clear. In early 2007, the mean number of iterations per ticket was of about 1.4, while in 2012 it remains around 1.8 most of the year.

Despite this increase in the number of iterations, the mean delay for tickets has been decreasing since mid 2008, after a couple of long peaks (in mid 2007

and mid 2008), for which we have found no apparent explanation. Looking at the general trend, it shows how, despite putting more effort in the review process (more iterations), the project is being able of reducing the time-to-review. This means that both reviewers are being more responsive to review requests by authors, but also that those have into account quickly the suggestions for changes, so that a new review cycle can start.

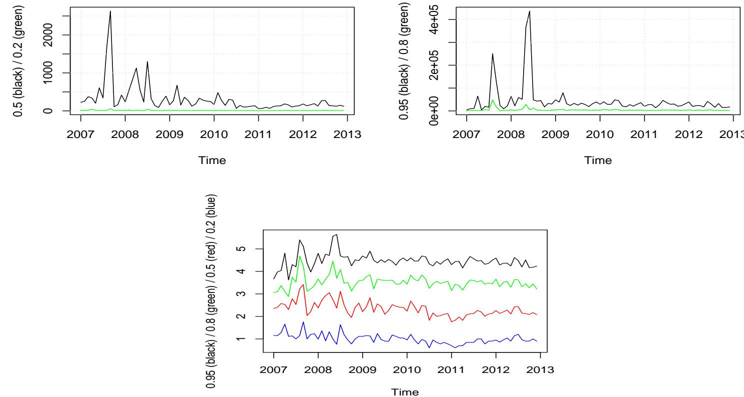


**Fig. 4.** Performance parameters (quantiles per month). Delay (time from review request to review approval, in minutes) for "quick" (top) and "slow" (middle) reviews, and logarithm of the delay (bottom).

This said, it is important to signal that the distribution of delays is very skewed: while the median for delays is 151 minutes, the mean is 7,447 minutes. Therefore, we have considered convenient to offer also, in Figure 4, similar information, but now using quantiles. In the top chart in that figure we can see the maximum delay for the quickest closed tickets. For example, the .2 (green) line in that chart shows how the maximum delay for the 20% quickest review processes, over time. The .5 (black) line shows the evolution of the median delay.

For all the quantiles analyzed, the evolution of delay over time is quite similar, and consistent with the one found for the mean delay. Maybe the quickest tickets are reducing their delays, while the slower ones tend to be more stable. This can be seen in the middle chart, with the delays for 80% and 95% of the tickets, but more clearly in the bottom one, which shows the logarithm of delay over time: the red and blue lines shows a tendency to descend since 2008, while the black and green ones are almost horizontal. The bottom chart, taking into account the log scale, shows also the great skewness of the distribution of delays.

## 5 Discussion and threats to validity

In large projects where many different actors contribute, each with their own and usually competing interests, many software development processes are difficult, yet important to understand. In the case of our study, the code review process is specially important, because it controls what enters the code base, but also what is left out. It is a barrier that developers have to overcome for contributing. Therefore, understanding it is really important, and more when the stakeholders are companies competing in the marketplace, but collaborating in the project.

A pure qualitative understanding, based on the modeling of the mechanics of the process, is not enough. Quantitative information is needed to back discussions with data, to detect early problems, and to be able of evaluating solutions and new policies. In this respect, we have quantified several aspects of the process, and have validated them with WebKit developers.

The main contribution of our study, from this point of view, is a detailed methodology, that can be used in the WebKit project and, with some variations, in other projects too. The parameters and charts presented can be the basis for a specialized dashboard that tracks the details of how the review process is evolving. The parameters presented, and the way they are calculated, can also be the basis of a validation system for any quantitative model of the code review process.

There are several threats to the internal validity of the study. The main one is probably the validity of the parameters selected to characterize the code review process. In general, both practitioners and academics consulted agree on the validity and usefulness of them as they can be linked to important concepts such as effort or delay in actions. But more research is needed to really correlate them with other parameters, so that it becomes clear that they are really important for the review process. Other threats to internal validity are related to the actual data retrieval process, the validity of the analyzed sample, and the exact procedures for estimating the parameters. In general, all of them have been validated with developers from the project. Section 3, and the answer to Q1, have also tried to establish how the sample is good and large enough, as well as the process for estimating parameters from it. However, errors and conceptual problems may remain.

With respect to external validity, it is important to notice that this study does not try to state proprieties to be valid in other projects, nor even in the future of WebKit. We have only tried to determine techniques and artifacts that help to understand the review process, and not to determine general laws or models of how it works. This said, the methodology and the presented artifacts (charts, statistics) are meant to be valid for other projects, and therefore threats to external validity can be applied to them.

## 6 Conclusions

This paper has presented a detailed methodology for the quantitative analysis of the code review process in large software development projects, based on traces left in software repositories. The methodology has been tested with WebKit, a large and complex project with high corporate involvement. Some developers have given us assistance in the validation and understanding of the code review process, ensuring a higher usability of the results for its stakeholders.

We have answered the two research questions stated in the introduction of this paper. First of all, we have determined how there is enough information in the project repositories to characterize the code review process, and how it can be used, in fact, to calculate parameters that seem to be related to the extension and performance of the project (Q1). We have also characterized the evolution over time of the process using quantitative (bulk or performance) parameters, and have shown how they can be useful to understand such evolution (Q2).

By doing this we have characterized the review process of the WebKit project, and proposed the fundamentals for a dashboard that can serve to evaluate it. We have found that the importance and extension of code review is growing in the project, that reviewers are not growing as fast as authors - and although this has not supposed delays so far, it could cause bottlenecks in the future. On the contrary, the project is improving in the code review process over time, probably due to developers devoting more effort to it.

## Acknowledgments

The work of Gonzalez, Robles and Izquierdo has been funded in part by the Spanish Gov. under SobreSale (TIN2011-28110) and Torres Quevedo (PTQ-12-05577). We thank the Webkit developers for their feedback and suggestions.

## References

1. A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. Software inspections: An effective verification process. *Software, IEEE*, 6(3):31–36, 1989.
2. A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40. Elsevier Inc., 1984.
3. Jesus M. Gonzalez-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17:75–89, 2012.
4. P.C. Rigby, D.M. German, and M.A. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.
5. P.C. Rigby and M.A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.