Using Metrics to Track Code Review Performance

Daniel Izquierdo-Cortazar
Bitergia
dizquierdo@bitergia.com

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

ABSTRACT
During 2015, some members of the Xen Project Advisory Board became worried about the performance of their code review process. The Xen Project is a free, open source software project developing one of the most popular virtualization platforms in the industry. They use a pre-commit peer review process similar to that in the Linux kernel, based on email messages. They had observed a large increase over time in the number of messages related to code review, and were worried about how this could be a signal of problems with their code review process.

To address these concerns, we designed and conducted, with their continuous feedback, a detailed analysis focused on finding these problems, if any. During the study, we dealt with the methodological problems of Linux-like code review, and with the deeper issue of finding metrics that could uncover the problems they were worried about. For having a benchmark, we run the same analysis on a similar project, which uses very similar code review practices: the Linux Netdev (Netdev) project. As a result, we learned how in fact the Xen Project had some problems, but at the moment of the analysis those were already under control. We found as well how different the Xen and Netdev projects were behaving with respect to code review performance, despite being so similar from many points of view.

In this paper we show the results of both analyses, and propose a comprehensive methodology, fully automated, to study Linux-style code review. We discuss also the problems of getting significant metrics to track improvements or detect problems in this kind of code review.

KEYWORDS
Software development analytics, Code review, Data mining

1 INTRODUCTION
Pre-commit peer code review is an increasingly common practice in software development. It helps to maintain long term software quality of large and long lived projects [7] [6] [1], and it is also known to increase team awareness, knowledge sharing and creation of alternative solution to problems [1]. To that end researchers have attempted to understand the successes, challenges and factors that impact the code review process [6], [1], [15], [9].

Despite all this attention, there is a lack of documented cases where software development teams have characterized their code review practices, or of research driven by the directly stated needs of those teams. This paper is one of those cases: the research questions were designed in collaboration with practitioners, and the results were shared and refined with them. This allowed the team to learn about their own practices, and to integrate metrics in their understanding of their own development process.

The study presented in this paper was first commissioned by the Xen Project Advisory Board. Xen is devoted to produce a software virtualization platform which is in use on some of the largest clouds worldwide. It uses pre-commit peer review, based on messages in mailing lists, following a review process quite similar to that of the Linux kernel [5]: the “Linux-style code review”. In summary, proposed changes are sent to a mailing list where they are discussed. If after the discussion new versions of the proposed changes are requested, they are again sent to the mailing list. When at some point, proposed changes are deemed acceptable, they get committed to the code base. Anyone on the mailing list can comment on patches, but only some of them can decide that they are ready to be accepted and committed.

In mid 2015, some people in Xen Project Advisory Board had observed an increasing trend on the number of review messages in their mailing list, and became concerned about the performance of their code review process. They had a perception that maybe effort devoted to code review was increasing as well, and over all, that time from submission of a patch to commit in the code base was getting increasingly
long. Due to peculiarities of their process, it was not easy to have metrics that could help them to understand what was really happening. In fact, it was even difficult to define which evidences could be used to decide if their code review process was getting longer and more effort-consuming. Therefore, we conducted a study which started by finding out what metrics to use, and how to compute them. The preliminary results [4] showed some of these metrics, and how the project seemed to have them under control.

A more complete study followed, which included tracking the review process during a longer period of time, refining the metrics used, producing a dashboard for continuous monitoring³, and using a comparable project for benchmarking the results. The selected comparable project was Linux Netdev, which develops the part of the Linux kernel related to networking support. The results were validated by practitioners in the Xen project, and their feedback was used to refine further the methodology. This paper presents the results of all this process.

The research questions explored by the complete study can be summarized as follows:

- **RQ1**: How can we characterize the overall code review activity?
- **RQ2**: How can we characterize the impact of code review in delaying the adoption of proposed changes?
- **RQ3**: How can we characterize the impact of the complexity of proposed changes in that delay?

All three questions were targeted to answer the practical questions of interest to the project: was their code review process having an increasingly negative impact in their time to adopt changes, and were they devoting increasingly more effort to it?

Our study shows that despite being comparable in many respects, including their code review practices, both projects have different answers to some of those practical questions. Both have a review activity in the same order of magnitude, but while the Xen project has delays due to code review under control, it is increasing very quickly in the case of Netdev project. This information has helped Xen to make some decisions to further improve their code review, and to have a continuous monitoring platform which they are using to continuously tracking the process².

The main contributions of our study are:

- We show that it is possible in practice to employ software development analytics to understand problems in code review processes.
- We propose a quantitative methodology for analyzing “Linux-like” code review process. It is completely automated, reproducible, and could be used in other cases with similar code review processes.
- We provide open access to all the data produced, which has shown to be useful for informed discussion, and for letting the analyzed projects better understand and improve their code review processes.

The rest of the paper is structured as follows: Section 2 presents the two projects we have analyzed. Section 3 presents the methodology. Section 4 details the quantitative results of our analysis. Section 5 presents the main threats to their validity. Section 6 discusses results and their relationship with related work. The paper concludes with Section 7 where we present conclusions and future work.

2 XEN, NETDEV AND THEIR REVIEW PROCESSES

The Xen project³ develops and maintains the Xen hypervisor, one of the key components in some of the largest cloud services and cloud-related products, such Amazon Web Services, Oracle VM Server, IBM Softlayer, or Rackspace Public Cloud. In this paper we will refer both to the software it produces, and to the project itself, as “Xen”. Although Xen is an independent project, it needs coordination with the Linux project in many areas.

Linux Netdev is a subsystem of the Linux kernel, which handles all its network-related aspects. It includes core code for network drivers, network security, and network mobility. In this paper we will refer both to the software it produces, and to the project itself, as “Netdev”. As it is common in the hierarchical structure of Linux development, Netdev has a certain level of autonomy, but is subject to coordination with the rest of the Linux project.

Both Xen and Netdev are used in many different environments, and therefore they need to be continuously updated to fit new requirements, and to fix bugs that may appear. Due to the kind of software they produce, and its use in mission-critical applications, both are pressed to release often, with as much quality as possible. In many cases they are used in platforms using continuous deployment practices, which want frequent access to the latest changes to their code base. Being as quick as possible in fixing bugs, and in letting new features enter this code base are key requirements in both cases.

Xen and Netdev are projects producing free, open source software (FOSS), which are run as open development projects (or “open source style software development” [8]) with all their development information shared publicly. In both cases, developers working for many different companies collaborate together to maintain that code base. Netdev, being a part of the Linux kernel, follows its software development practices. Xen, for historical and practical reasons, follows similar practices as well.

Both projects conduct code review in public mailing lists. They use review threads for each proposed change, which can be automatically identified from the general traffic in the list. When a proposed change is finally approved, it gets committed to the git repository. The first line in the commit comment is similar (but not always equal) to the subject in the email thread.

The following terminology is used for proposed changes:

---

1. http://xen.biterg.io
2. http://xen.biterg.io
3. http://www.xenproject.org/
Patch: A “patch” is the basic unit of change. It includes the description of a code change (lines added, lines removed), and some related information about it. A patch can be a fix to existing code (for example, to fix a bug), or a some code being developed for future releases (for example, to support a new feature).

Patch series: A combination of one or more patches that are interdependent, and should be reviewed together, is a “patch series”. Developers decide when a set of patches should be considered as a “patch series”, usually attending to the fact that either all patches should be accepted (committed) together, and at the same time, or none should. In this paper we will refer to a standalone patch as a patch series of one patch.

Version: Each patch series can be resubmitted several times, to address comments by reviewers. Each time it is resubmitted, it is considered as a new version of the patch series. Versions are tagged with consecutive natural numbers, starting at one.

Merging a patch series: When a patch series is committed to the code base of the project. All patches in the patch series are committed in an atomic operation. Each patch is committed as one commit.

Developers usually work in a patch series until they consider it ready for review. At that point, the code review process begins, when they post the patch series to the mailing list. They send each patch in one message and, if the series includes more than one patch, another message for the patch series itself. All the messages for a patch series has the same subject, except for an identifier for the patch (a natural number) and a version (only if the version is not the first one). Peer developers review the proposed changes for quality and functionality by submitting their comments to those messages.

When some of the review comments are not positive, it is considered that the patch series is “rejected”. In this case, the developer has to address those comments with changes to the patch series. Once those changes are ready, the patch series is resubmitted as a new “version”. When, usually after several versions, reviewers agree that the patch series meets the required quality and functionality, it is “accepted” to be included in the code base. This means that it is “merged” (committed) to the git repository.

All this process, since when a patch series is posted for first time to the mailing list, until when it is accepted to be committed is what we will refer to as the “code review process” for that specific patch series. The full process is described for developers in the Linux kernel documentation, section “Submitting Patches”.

Table 1 shows some parameters characterizing the volume of activity related to code review in both projects. It can be seen how not only both projects are very similar, but also their code review activity is in the same order of magnitude. This is the reason why Netdev was selected as a benchmark to put in context the parameters in Xen, providing an interesting opportunity for learning from the similarities and differences in the evolution that we select to characterize their code review processes.

### 3 METHODOLOGY

The methodology we used for the study described in this paper is the same for Xen and Netdev, and it is based on the quantitative analysis of mail messages related to code review in public mailing lists, and commits in public git repositories. A part of this methodology was presented in [4], which refers to a preliminary version of it, that we conducted only on Xen. Then, we refined it with the help of the Xen community, and used it to analyze both Xen and Netdev. The analysis was conducted for code review processes during the period 2012-2015.

The methodology is structured in several phases:

- Data retrieval. Retrieval, parsing and conversion of unstructured code review messages (patches and comments) found in mailing list archives, and commits found in git repositories, into organized information in MySQL databases (Subsection 3.1).
- Data matching. Matching patches to corresponding commits (Subsection 3.2). This is important to be able of computing how long does it take since a patch is proposed until it is merged in the code base.
- Visualizations and analysis of the resulting dataset (Subsection 3.3) to show metrics of interest, characterizing the code review process.

For both cases, we run the same methodology, starting from the data sources corresponding to the analyzed project. For Xen we analyzed; mailing list: xen-devel\(^{9}\) and git repositories: Xen\(^{6}\), Mini-os\(^{7}\), Raisin\(^{8}\), and Osstest\(^{9}\), which include most of the patches reviewed by the Xen project. And for Netdev we analyzed; mailing list: linux-netdev\(^{10}\) and Torvalds’ Linux kernel git repository\(^{11}\), which contains all accepted patches for the Linux kernel, including those for Netdev.

Each phase is described in detail in the next subsections.

#### 3.1 Data retrieval

In summary, this phase consists in running CVSAnalY and MLStats on the data sources for each analyzed project (see Figure 1).
Step 1: Retrieval of mailing list data. We use MLStats (Mailing Lists Stats) [13] to retrieve mail messages from the development mailing lists. MLStats produces a structured MySQL database, from which we use the following tables:
- mailing_lists: metadata about mailing lists analyzed.
- messages: metadata for messages, including contents, and classification in threads.
- people: people involved in messages.
- messages_people: relationship between people and messages.

Step 2: Retrieval of Git information. We use CVSAnalY [13], a tool that retrieves and organizes data from source code management systems and stores it in a MySQL database. From it, we use the following tables:
- scmlog: general metadata for each commit.
- actions: actions performed by each commit on each file.
- file_types: kinds of files.
- branches: distinct branches.
- people: name and email of people who contributed to the repository.
- repositories: metadata for analyzed repositories.

3.2 Data matching
This phase consists on the identification of messages related to coded review, their grouping according to the corresponding patch and patch series, and their matching to commits in the git repository. We developed a script (patch_matcher.py) to produce a database with the resulting match (see Figure 2).

Step 3: Detection and classification of messages related to code review. To select messages related to code review we filter, from the MLStats database, those with the key word “PATCH” in the subject. The usual pattern found as subject in those cases is:

```
PATCH[version>]<patch number nn/mm<subsys:><headline>
```

“PATCH” is the keyword signaling that this message corresponds to the code review for a patch. Then we have the version of the patch, if it is not the first version. Then we see the ordinal number of the patch (“nn”) with respect to the total number of patches (“mm”) in the patch series. This part may be omitted in patch series composed of just one patch, although in some cases we can see “1/1” as well. “subsys” is the name of the subsystems affected by the patch, and “headline” is the name of the patch, which usually corresponds to the first line in the commit message, when the patch is finally committed. The first message in a patch series starts a new thread, with all its patches, and review to patches, found as replies to it. A new version of a patch series starts a new thread with the same headline.

When refining our heuristics to match as much patches as possible, we found some variations to this pattern. For example, we found cases where each patch started a new thread, others with missing or irregular numbering of patches, patch versions, or patch series. We also found different patches with identical subject messages, subject lines with missing version, patches with version starting at 4 instead of 1 or 1, and impossible time stamps. We used regular expressions and heuristics to identify messages for code review as accurately as possible. During this step we also parse some tags in message headers, which provide more information about the review status of the patch in the code review process: From, Signed-off-by, Tested-by, Reported-by, Reviewed-by and Cc.

Step 4: Merging information from mailing discussions and git activity databases. Once headlines are identified in relevant messages, we can match patches in mail discussion to their corresponding commits by comparing them to the first line in commit comments. According to available documentation, this is a reliable way of matching messages to commits. With the help of Xen developers, we refined from plain matching to a more complex heuristics which ignores spaces, considers irregularities in threads, considers some characters as equivalent, etc. That increased the fraction of matched messages, while not including false positives (we checked manually for it).

The code review process database that is produced as a result of this step includes the following tables:
- patch_series: information about each detected patch serie.
- patch_serie_versions: information about each of the versions for each patch series.
- patches: information about all of patches.
- comments: comments sent as a part of the review of a patch.
- people: people involved in the process.
Using Metrics to Track Code Review Performance

EASE’17, June 15-16, 2017, Karlskrona, Sweden

Figure 3: Production of metrics, and visualization.

- tags: tags found in each comment.
- commits: information about all commits.

Table 2 shows the raw numbers of information in the consolidated databases: number of patches identified in the mailing list and number of those patches that were linked to corresponding commits in the git repository.

Table 2: Number of patches, and number of patches linked to commits in the code review database (top: Xen, bottom: Netdev)

| Year | No. of patches | Patches matched to commits |
|------|----------------|---------------------------|
| 2012 | 1907           | 1108                      |
| 2013 | 2503           | 1549                      |
| 2014 | 2092           | 1558                      |
| 2015 | 2773           | 2114                      |

For the period 2013–2015 the fraction of patches linked to commits is about 70% for Xen and 63% for Netdev. A sample of matched commits was analyzed manually by the authors, in collaboration with Xen developers, to ensure that no apparent bias was present.

3.3 Data analysis and visualization

**Step 5: Analysis and visualization of relevant metrics.** We query the code review process database to compute the metrics that will be used to answer the research questions, and in the end, to assess the projects about how effective it is their code review process. For producing the results presented in this paper, we used a Jupyter notebook based on Python and Pandas. The notebook was refined several times, using the feedback provided by the Xen community.

For defining the metrics, we followed a GQM (goal–question–metric) approach [2]. The defined goals, and the metrics proposed for each of them, are as follows:

**Goal 1:** Find out the evolution over time of the review activity (corresponds to RQ1) and the over all effort that goes into a single code review. This is important to put other metrics in context. To derive questions and metrics for this goal, we decided to focus on the evolution of number of patches per patch series of 1-patch to 5-patches, and the average and median effort that goes into a single code review.

**Metrics:**
- Evolution of number of patches per patch series: as they are identified in the `patch_series` table in the code review database.
- Evolution of comments received per patch series (patch): The average and mean number of mail messages in the thread corresponding to a patch series or a single code review.

We measure comments as the number of review discussions received by a patch series, and a patch series as the number of combined patches (one or more patches) that are interdependent. Sufficient discussion is a good indicator of a high code review participation and quality [7]. Particularly in the Linux kernel patches that have received insufficient discussion may find it difficult to get into the next kernel releases [5].

**Goal 2:** Find out if the delays imposed by code review are increasing or decreasing (corresponds to RQ2). We decided to focus on time-to-merge (time from the proposal of a patch series to its merging in the code base), as it captures the complete delay due to code review. This time very important in continuous deployment scenarios, since proposed changes can be deployed only once they are reviewed and merged.

**Metrics:**
- Time to merge for patch series merged.

**Goal 3:** Find out if there is an impact of the complexity of patch series in how long they take to be reviewed (corresponds to RQ3). Xen developers considered that the main factor affecting complexity for review was the size of a patch series, computed as the number of independent patches in it. Since each patch (per definition) deals with a specific issue, but all of them implement a coordinated change, the more patches the more interrelationships, and the more overall complexity.

**Metrics:**
- Time to merge for patch series split by number of patches in the patch series.

The goals that the Xen project had were adapted as research questions of the study, so that the results can be generalized to characterize projects. The derivation of questions and metrics from the goals was done in collaboration with Xen representatives, in an iterative process.

4 QUANTITATIVE RESULTS

This section will show the results of applying the methodology to Xen and Netdev. Although the study was focused on Xen, the results for Netdev provide an interesting benchmark for comparison.

4.1 Evolution of comments and patch series

The data about review comments and patch series answers RQ1. We use the evolution of these metrics over time to characterize the overall code review activity and the effort...
that goes into a single code review, since they were considered the most useful parameters by the Xen project.

Figure 4 shows the evolution of number of patches per patch series of 1-patch to 5-patches. In both cases, the relative evolution of patch series of 1-patch is similar. However in the first semester (first six months) of 2014 and 2015, Xen exhibits large patch series of 5-patches as compared to Netdev where the evolution of patch series of 5-patches has remained relatively constant for the period of study.

Figure 5 shows the average and median effort that goes into a single code review measured as number of comments received by a single patch for the period of study. For Xen, there is an increase in the average number of comments per patch especially during the first semester of 2014 and 2015, confirming the perception of Xen developers of a larger review activity during this period, which originated this study. While for Netdev, on the contrary the average number comments per patch is gradually decreasing, which shows that the review activity is declining. This signals little effort is invested in the code review activity which may lead delays in the code review process.

From Figure 4 and 5 the large size patch series of 5-patches during first semester of 2014 and 2015 at Xen generated long discussion in terms of review comments per patch with an average of 15 comments which could explain the perception of a complex review activity by Xen developers during this period.

4.2 Time–to–Merge

Time–to–merge answers RQ2, which we measure as the time since the proposal of a change (post of a new patch series to the mailing list) to the merging in the code base (patches committed to the git repository). Xen agreed that this was the most important parameter to track when considering delays imposed by the review process. From a more general point of view, it is known that short merge cycles motivate developers to be more productive [6], and lead to early identification of bugs and fast deployments [14].

Figure 6 represents the evolution of time–to–merge over time. It shows how time–to–merge in Xen was increasing slowly since 2012 until the first semester of 2014, when it started to decline. During the last year, not only the median (which is now well below 5 days), but also the spread of time–to–merge has been reduced. This is interpreted by the project as a positive evolution.

However, Netdev shows a very different story. In the period 2012–2014, time–to–merge was under control, and small. But during the last semesters, it has been increasing very quickly. Although the median is still relatively small (but higher), slow reviews are really long, with 25% of them taking longer than one year to review.
The data shown in these charts addresses the main concern for Xen. They can track when the situation is getting out of control, and provide a simple metric to improve. They also illustrate that there is nothing specific in the way they review that maintains time–to–merge down, because the benchmarking project shows a very different evolution, even being similar in many ways. Now they are using these metrics to evaluate the measures they are putting in place to keep time-to-merge as low as possible.

To explain the differences in time-to-merge between Xen and Netdev, we looked at other metrics. Among them, the number of reviewers, which we tracked using the "Reviewed–by" tag. During 2015 Xen had 103 reviewers, for a total of 2,114 patch series merged, while Netdev had 69 reviewers for 2,057 patch series merged. This would mean Netdev reviewers were too overworked to keep time–to–merge low.

The evolution in Xen explains how they kept review delays under control: in 2014 they had only 69 reviewers: the increment in reviewers could cope with the pending review work.

4.3 Impact of patch series complexity

Figure 7 represents time–to–merge of patch series, classified by number of patches per patch series. Xen developers considered this metric (number of patches) has a clear impact on the difficulty of the review. When a patch series is composed by many patches, each of them has to be reviewed independently, and the relationships between them all, and with other parts of the code, too. Therefore Xen developers had a perception that there could be a negative impact of the number of patches per patch series on time–to–merge.

Metrics for both projects show that Xen developers were right. Except for the two–patches case, there is a clear increase in time–to–merge with the number of patches per patch series. Two–patch cases show an increase too, even higher than expected, compared to other cases. For example in Xen, time–to–merge for 75% patch series containing one patch was of about 10 days, while for four–patches patch series it is of about 70 days. These results encourage the project to keep the number of patches per patch series as low as possible, and to keep an eye on the evolution of patch series with a large number of patches. For this, we produced visualizations of the evolution of the number of patches per patch series represented in Figure 4 earlier in Section 4.1. Indeed the large patch series of 5-patches during first semester of 2014 and 2015 at Xen explain why time–to–merge (see Figure 6) was high this that period.

5 THREATS TO VALIDITY

This paper presents the final results of a study we have conducted for the Xen project. Some preliminary results, which are in line with those shown here, were presented in [4]. These results are subject to some threats to validity. In our case, since the main focus of the study was the specific situation in a given project (Xen), we were specially interested in highlighting factors threatening internal validity.

One of the key parts of the methodology is matching commits with patches in mailing lists. But even when we improved heuristics thanks to the feedback from the Xen
community, we could not attain 100% matching. We observed slight variations in the traces we analyze both in messages and commits, which we couldn’t always capture. Since not all steps in the code review process are automated (in fact, many of them have human intervention), the assumption that the headline of a commit message matches the subject of the corresponding patch, not always hold, or new versions of a patch could have a different subject in messages. This means that some code review processes are not adequately measured, or are not measured at all.

Another reason for the problems in matching are cross-posted patches from other projects. These patches never land in the Xen (or Netdev) git repository, because they are not intended for that: they are just cross-posted seeking review by related projects, as a way of coordination in case of changes that could affect related functionality.

“Linux-like” code review does not have a way to “abandon” a proposed patch. If a developer no longer wants to produce new versions of a patch, there is no way of signaling that in mailing lists, at least in a way that can be detected automatically. That means that there is no way of telling very long code review processes from abandoned processes. Fortunately, this will have little impact in our metrics, since we focus only on code review processes that were completed, with an accepted commit. But this means that the effort put in abandoned code reviews is not considered by our methodology, and that very long reviews are considered only once they are finished and committed, and therefore, ignored for a long period (while they are still active, attracting effort). These abandoned and very long reviews contribute as well to the imperfection of matching.

Despite these problems, the matching we attained was significant enough to draw conclusions, and we don’t have reasons to consider that the sample of code review processes that we could match is not significant enough, or is too skewed, to draw false conclusions. The impact of abandoned and very long review processes is not negligible, but fortunately small enough to influence seriously the results. In any case, due to them, both effort and time-to-merge could be underestimated.

With respect to external validity, we don’t claim that all project following code review should follow the same patterns that we have identified for the two cases we have studied. On the contrary, based on the differences found in those cases, it is clear that each project will have its own patterns. However, we propose the methodology as valid for all “Linux-like” code review practices, and the parameters computed, or other with similar semantics, as useful for any pre-commit peer code review process. Therefore, threats to external validity of these two results are important as well:

We have run our methodology in two projects which follow “Linux-like” code review, as defined in the Linux kernel documentation. Therefore, it should be possible to run it on any other project following the same practices. This includes all projects (or subprojects) producing components for the Linux kernel, and may others that for historical or practical reasons follow those practices. However, we have not studied in detail how strictly those projects follow the process, neither if they conduct all their code review using it. Some projects, could not be strict with code review, allowing for off-channel reviews, such as when one developer informally reviews the work of a peer, with no reflection in public channels. In some others, the source data could be very difficult to identify, for example if it is spread in many different mailing lists. He have only cursorily analyzed some Linux-related projects, finding that it is very likely that the methodology can be applied.

The parameters of interest in our study have been defined in a process which included the feedback from the Xen community. This is one of the key points ensuring that they are useful for them. But because of that, there is in principle no reason to think that they are useful for any other project. However, their concerns (effort devoted to code review, and delays imposed by the process) are shared by many other projects, and specially those working in areas where continuous deployment is usual, and where reviewers have to be found among experienced developers. Therefore, it would be more correct to say that the parameters of study are likely to be useful in projects that share those concerns. But in any case, a more complete study, with a larger sample of projects, should be conducted to assess on the usefulness of the proposed parameters for other projects.

Another external threat for the validity of the parameters is to which extent they can be computed from the available data sources. In the case of projects following “Linux-like” practices, that should be no problem (except for the discussion we already had for the first item in this list). But for others, the specific definition of the parameters should be revisited, and of course the way for computing them. Since the concepts used are common in pre-commit peer code review, it shouldn’t be difficult to find a way to define how to compute them. But the proposal should be tested on a number of projects with different practices, and using different tools to assist code review, to be sure of its validity.

6 DISCUSSION AND RELATED WORK

The metrics that we have selected for assessing Xen about the issues they wanted to track and for benchmarking the results at Netdev are not new. In fact, there is abundant previous research highlighting similar metrics:

Metrics for activity. Different authors have linked a high number of review discussion to software quality [12] [7] [6]. For the case of Xen, when activity is increasing (measured as the average number of comments per patch), would mean that the overall quality is similar. For Netdev, the same happens in a context of decreasing overall activity. Therefore these activity metrics provide an interesting context to the effort devoted to code review, time-to-merge metrics and the expected quality: the latter could be improved by adding more reviewers in the review activity.

Time-to-merge. It is clear that an effective and well-done code review should be done in a timely manner [6]. In the
case of Xen, this is parameter of specific interest, because the project is interested in keeping it as low as possible because of its impact on continuous release and continuous deployment downstream (in products derived from Xen).

Impact of patch series complexity on time-to-merge. Patch series complexity is of special interest, because there is a certain dispute, both in literature and among practitioners, about whether small (split) or large (combined) patches (in our case, patch series) are easy and quicker to review [3] [16] [11] [6]. Intuitively, large or combined patch series are more difficult to review because they touch more areas of code, and have more interrelationships. However, reviewing code in small, isolated chunks may be too difficult. We found such mixed perceptions on patch series complexity among practitioners in Xen, but that happens in other projects as well. For example, Mozilla developers have been quoted with mixed opinions [6]. Some other projects have specific recommendations, such as Linux, which advises developers to split patches to reduce complexity [5]. Based on the results of our study, it seems sensible at least in the case of Xen to follow this recommendation. However, patch series complexity is also a crucial factor when developers decide which patches to review [5], which could mean that smaller patch series are reviewed faster because reviewers start to work with them earlier, not because they are reviewed in less time.

In addition to the metrics of interest of the Xen project presented in this paper we computed other metrics that can lead to an informed discussion about the performance of a code review process, detailed in the reproduction package notebooks. These include; Time-to-commit, Time-to-rework, Cycle time, Reviews completed, active, stalled and ongoing, Added and removed lines for each review, People domain analysis among others.

There are some other practical aspects of our study that are worth mentioning:

Code review based on mailing list messages have been characterized as difficult to study due to the complexity involved in matching patches submitted for review on the mailing list to their corresponding committed patches in the git repository [10]. Mailing list review processes are also known to contain incomplete information which specialized tools cannot resolve perfectly [5]. We had to overcome this problem, at least partially, to produce meaningful results.

The main trouble with the study is finding the relevant events (submission of a new patch series, review of it, merge in the code base, etc), and link them to patch series and commits. When developers use specific tools to assist in code review, such as Gerrit, this is relatively simple to do. But when little tooling is used, and many actions are manual, as is the case, the task is more difficult, and prone to errors due to small variations in data, such as subtle changes to a “Subject” line in a message.

The technique used for matching patches to commits is based on the analysis of the subject in mail messages and the headline in git commits. A better technique could be matching the actual patch (changes to source code lines). However, we were advised by developers that our technique was good enough, so we stuck to it.

When assigning reviews to time periods we use the time for merging. This ensures that all the reviews we consider have had the opportunity of finishing, but has the negative effect that we do not consider code reviews that are still being processed. It also has the impact of artificially increasing time-to-merge for a period if during it a large number of old reviews are finished. That means that a high time-to-merge for a short period is not necessarily a bad signal: it usually means that the project is dealing with old patch series, allocating effort to review and merge them.

There is nothing in the methodology that prevents it being used in any project with Linux-like code review processes. In fact, we used the analysis for Netdev as a test for this hypothesis, with very positive results: we just retrieved its data sources, and run exactly the same tools, in the same way, that we had done for Xen.

The parameters we used for tracking code review were the result of our interaction with the Xen community. Even when they may be computed in different ways for other code review processes, their semantics are very likely relevant for any project with similar interests to Xen, namely, short time-to-production, to easy the release and deployment of downstream products derived from it.

After our interaction with Xen developers, we learned that the main advantage of defining tracking metrics was that they had data to back more productive discussions, leading to a more informed decision making about resource allocation or policies on development procedures.

Of course, our study is not the first one analyzing pre-commit peer code review: many researchers have attempted to understand its success, challenges and factors that impact in it [12] [9] [6] [1] [15].

Rigby and Storey [12] investigated the mechanisms that developers use to find code changes they are competent in projects with code review practice similar to those used in our case studies. They found that developers use email filters, progressive details within patches, bottom posting, recipients building and other techniques which are important to understand and evaluate the effectively of the code review process. Rigby et al. [11] [9] examined two peer review techniques: review-then-commit (RTC) and commit-then-review (CTR). Their results indicate that while CTR has a shorter review interval than RTC, the two techniques are likely to find the same number of defects. Our study focuses on RTC (pre-commit review) because that is the practice in Xen. However, it is nowadays a very common practice in FOSS projects.

McIntosh et al. [7] found a significant link between code review participation and code review coverage on software quality. Low code review coverage and participation was found to produce high post-release defects [7]. A similar study in a proprietary setting (Sony Mobile) [15] found a significant link between code review coverage and code review participation for externally developed code but integrated
with Sony mobile components. In the case of our study, all commits are reviewed, per the project policies. That would mean that its quality is comparatively high.

Rigby et al. [9] provides guidelines for conducting peer reviews: they should be frequent, incremental, synchronous. Particularly changes should be small, independent and complete to enable developers of different expertise to review changes with reduced communication and dependence [9]. Most of these practices are found to be followed in the Xen case.

7 CONCLUSIONS AND FUTURE WORK

Our main contribution relies on working with a specific project to define which aspects of the process they want to track and improve. Instead of trying to assess from a general point of view when code review is successful, we have helped to find out which specific metrics can provide an idea of when the process is improving or deteriorating, driven by the specific interests of the Xen project.

As a result of our study we have defined, in collaboration with the Xen project, metrics that help them to track aspects of the code review process that they consider important. We have produced software to compute those metrics in a completely automated way, and benchmarked the results and methodology to a similar project Linux Netdev.

In addition to the metrics of interest of the Xen project presented in this paper we computed other metrics that can lead to an informed discussion about the performance of a code review process, detailed in the reproduction package notebooks. These include: Time–to–commit, Time–to–rework, Cycle time, Reviews completed, active, stalled and ongoing, Added and removed lines for each review, People domain analysis among others.

Since the methodology and the software we have produced for this study are reusable for any “Linux-like” code review process, we intend to repeat the analysis on the whole Linux kernel, to learn how code review is applied in one of the largest and more relevant software systems. Many aspects of our methodology could be applied to other cases of pre–commit code review. We consider that it will be possible to work with other projects to extend it in a way that can help to track and compare relevant code review metrics for a large sample of cases.

ACKNOWLEDGMENTS AND REPRODUCTION PACKAGE

Daniel Izquierdo is partially funded by the Torres Quevedo program of the Spanish Government. Jesus M. Gonzalez-Barahona is partially funded by the Spanish Government, through project TIN2014-59400-R. Nelson Sekitoleko is funded by the Marie Skodowska-Curie - Research Fellowship Programme of the European Union, through the Seneca Consortium, grant agreement 642954.

There is a reproduction package12 with the data sources and source code for producing the results presented in this paper.

REFERENCES

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 712–721.
[2] Victor R Basili. 1993. Applying the Goal/Question/Metric paradigm in the experience factory. Software Quality Assurance and Measurement: A Worldwide Perspective (1993), 21–44.
[3] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2015. Investigating technical and non-technical factors influencing modern code review. Empirical Software Engineering (2015), 1–26.
[4] Daniel Izquierdo-Cortazar, Lars Kurth, Jesus M Gonzalez-Barahona, Santiago Dueñas, and Nelson Sekitoleko. 2016. Characterization of the Xen project code review process: an experience report. In Proceedings of the 12th International Workshop on Mining Software Repositories. ACM, 386–390.
[5] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast?: Case study on the linux kernel. In Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 101–110.
[6] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. In Proceedings of the 38th International Conference on Software Engineering (ICSE ’16). ACM, New York, NY, USA, 1028–1038. DOI: http://dx.doi.org/10.1145/2884781.2884840.
[7] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2015. An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering (2015), 1–44.
[8] Audris Mockus, Roy T Fielding, and James D Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology (TOSEM) 11, 3 (2002), 309–346.
[9] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary peer review in action: Lessons from open source development. IEEE software 29, 6 (2012), 56–61.
[10] Peter C Rigby and Daniel M German. 2006. A preliminary examination of code review processes in open source projects. Technical Report. Technical Report DCS-305-IR, University of Victoria.
[11] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. 2008. Open source software peer review practices: a case study of the Apache server. In Proceedings of the 30th International Conference on Software Engineering. ACM, 541–550.
[12] Peter C Rigby and Margaret-Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In Proceedings of the 33rd International Conference on Software Engineering. ACM, 541–550.
[13] Gregorio Robles, Jesus M Gonzalez-Barahona, Daniel Izquierdo-Cortazar, and Israel Herraez. 2009. Tools for the study of the usual data sources found in libre software projects. International Journal of Open Source Software and Processes 1, 1 (2009), 24–45.
[14] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In Proceedings of the 38th International Conference on Software Engineering Companion. ACM, 21–30.
[15] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, Ahmed E Hassan, and Naoyasu Uhayashi. 2016. A study of the quality–impacting practices of modern code review at Sony mobile. In Proceedings of the 38th International Conference on Software Engineering Companion. ACM, 212–221.
[16] Peter Weißgerber, Daniel Neu, and Stephan Diehl. 2008. Small patches get in!: In Proceedings of the 2008 international working conference on Mining software repositories. ACM, 67–76.

12https://gsyc.urjc.es/~jgb/repro/2017-case-xen/