CROP: Linking Code Reviews to Source Code Changes

Matheus Paixao  
University College London  
London, United Kingdom  
matheus.paixao.14@ucl.ac.uk

Jens Krinke  
University College London  
London, United Kingdom  
jens.krinke@ucl.ac.uk

Donggyun Han  
University College London  
London, United Kingdom  
d.han.14@ucl.ac.uk

Mark Harman  
Facebook and University College London  
London, United Kingdom  
mark.harman@ucl.ac.uk

ABSTRACT
Code review has been widely adopted by both industrial and open source software development communities. Research in code review is highly dependent on real-world data, and although existing researchers have attempted to provide code review datasets, there is still no dataset that links code reviews with complete versions of the system’s code base mainly because reviewed versions are not kept in the system’s version control repository. Thus, we present CROP, the Code Review Open Platform, the first curated code review repository that links review data with isolated complete versions (snapshots) of the source code at the time of review. CROP currently provides data for 8 software systems, 48,975 reviews and 112,617 patches, including versions of the systems that are inaccessible in the systems’ original repositories. Moreover, CROP is extensible, and it will be continuously curated and extended.

CCS CONCEPTS
• Software and its engineering → Software libraries and repositories;

KEYWORDS
Code Review, Repository, Platform, Software Change Analysis

ACM Reference Format:

1 INTRODUCTION
In software development, code review is an asynchronous process in which changes proposed by developers are peer reviewed by other developers before being incorporated into the system [1]. The modern code review process has been empirically observed to successfully assist developers in finding defects [3, 10], transferring knowledge [1, 16] and improving the general quality of a software system. Given its benefits, code review has been widely adopted by both industrial and open source software development communities. For example, large organisations such as Google and Facebook use code review systems on a daily basis [5, 9].

In addition to its increasing popularity among practitioners, code review has also drawn the attention of software engineering researchers. There have been empirical studies on the effect of code review on many aspects of software engineering, including software quality [11, 12], review automation [2], and automated reviewer recommendation [20]. Recently, other research areas in software engineering have leveraged the data generated during code review to expand previously limited datasets and to perform empirical studies. As an example, in a recent study we used code review data to analyse whether developers are aware of the architectural impact of their changes [14].

Code review research relies heavily on data mining. In this context, some researchers have attempted to mine code review data and have made their datasets available for the community [6, 7, 13, 19]. However, code review data is not straightforward to mine (see Section 2.2), mostly due to difficulties in linking the reviews to their respective source code changes in the repository. This limits the potential research that can be carried out using existing code review datasets. In fact, to the best of our knowledge, there is no curated code review dataset that identifies and provides the complete state of the system’s source code associated with a set of code reviews.

Based on this observation, we introduce CROP, the Code Review Open Platform: a curated open source repository of code review data¹ that provides, not only the review’s metadata like existing datasets, but also links, to each code review, a complete state of the system’s source code at the time of review. For each code review in CROP, one will be able to access the source code that represents the complete state of the system when the review was carried out. Thus, researchers will now have the opportunity to analyse code review data in combination with, for example, source code analysis performed by static and dynamic techniques such as profiling, testing and building. The combination of code review data and source code analysis will facilitate research in areas that previously required a significant amount of human participation, as outlined in Section 4.

Gerrit [15] is a popular open source code review tool that has been widely used in research [4, 11, 14, 19]. In addition, notable open source organisations adopted Gerrit as their code review tool, including Eclipse, OpenStack and Couchbase. Thus, since CROP focuses on curating code review data from open source software

¹https://crop-repo.github.io/
we have mined and linked 225,234 complete source code versions. The Gerrit system is built on top of git, and its code review process is outlined in Figure 1. As previously mentioned, Gerrit is built on top of git. Thus, the original code base in the repository and submitting a new review to the system’s repository can be reviewed. However, linking the reviews to changes in the system’s source code is outlined in Figure 1. A developer starts a review by modifying the original code base and making changes. The Gerrit system provides RESTful APIs that one can use to access the review’s data. For each new review, Gerrit creates a ChangeId to be used as an identifier for that review during its reviewing cycle. Other developers of the system will serve as reviewers by inspecting the submitted source code and providing feedback in the form of comments. Patches that improve upon the current revision are submitted as new revisions of the same commit. Finally, the latest revision is either merged or abandoned, where the first indicates the review was incorporated to the system and the latter indicates the commit was rejected.

2.2 Challenges in Mining Gerrit Review Data

Gerrit provides RESTful APIs that one can use to access the review’s metadata for a project, such as author, description, comments etc. However, linking the reviews to changes in the system’s source code is far from straightforward.

As previously mentioned, Gerrit is built on top of git. Thus, the git repository of the system would be the obvious first choice to access the versions of the source code that correspond to the code reviews. However, the system’s git repository is an unreliable record because Gerrit constantly rewrites and deletes history information.

When a new review is submitted by a developer, Gerrit creates a temporary branch in the git repository to be used for review. Every improved revision submitted by a developer is committed to this branch and replaces the previous review through a commit amend operation. Therefore, given a merged review, the review’s revision history is lost and only the source code of the latest accepted revision can be accessed. Moreover, when developers opt to abandon a review, the current revision is simply deleted from the repository.

In addition to the issues of lost history described above, the system’s git repository might also contain inconsistencies if we fail to fully account for the overall review process: code review is a laborious task, and it is common for some reviews to take a few days to complete one iteration of the core cycle [8, 17, 18]. Between the time a comment is initially submitted and the time the revision is finally merged to the system’s repository, other developers might have merged and/or committed other changes to the repository. In this case, each new revision submitted during the review needs to be rebased to be up-to-date. Thus, when one reverts the system back to the merged review, the source code will reflect not only the changes due to one but also all the other changes that were merged to the repository while the revision was open. These difficulties in isolating the source code changes associated with a specific review pose serious threats to the validity of empirical studies that use code review data.

2.3 Mining Code Review Data From Gerrit

We performed a preliminary analysis of the different open source communities that adopted Gerrit and selected the data source we would use for CROP’s development. As a result, we identified the Eclipse and Couchbase communities as those that provided all the data we needed to build CROP. The data mining process we employed is outlined in Figure 2.

As one can see from the figure, our mining framework consists of 4 sequential phases. The framework is written in Python, and we made it available online. Given a certain Eclipse or Couchbase project, the review harvester explores Gerrit’s RESTful API to download the code reviews’ metadata for the project. The API returns the data in JSON files that are kept to be used later.

In Phase 2, the snapshot harvester downloads the complete source code of the project for each code review. Both the Eclipse and Couchbase communities have a server that is separated from their git repositories and the Gerrit system where they keep complete snapshots of their projects for every commit ever performed in the project. These snapshots include the complete code base, i.e., source code, testing code, build files and so on. Thus, for each review, we iterate over all revisions and download the project’s snapshots that correspond to the code base both before and after each revision.

As a result of this process, we were able access versions of the project’s code base that would otherwise have been lost in the official git repository, such as reviews that were abandoned and intermediate revisions that were submitted during the review process. Moreover, by downloading the before and after versions of the code base for each revision, we check guarantee that the observed changes in the code were specifically attributable to the revision.

After downloading all the code reviews’ metadata and the project’s snapshots (Phases 1 and 2), Phases 3 and 4 handle the data. The discussion grinder processes the code review data stored in the JSON
We provide a URL we created a new git repository to store the project’s code base for each revision. The source code for each revision were committed sequentially, has a single master branch, where the before and after versions of the code base as it was after the revision was submitted. After_commit_id similarly, one should refer to the project's code base as it was before the revision was submitted. Before_commit_id the code base communities. Table 1 reports statistics concerning the data collected for each of these 8 systems, where the Eclipse projects are presented in the upper section of the table and the Couchbase projects in the lower section. As one can see from the table, all projects have more than 3.5 years of reviewing history, where the data for eGit spans more than 8 years. In addition, each project has more than 3,000 reviews and more than 8,000 revisions. In total, CROP provides comprehensive code review data linked to versions of the code base for 48,975 reviews and 112,617 revisions. Finally, these 8 projects are developed in a wide range of programming languages that include Java, C++, JavaScript, Python, Go and others.

Table 1: Statistics about each project currently in CROP

<table>
<thead>
<tr>
<th>Systems</th>
<th>Time Span</th>
<th>#Reviews</th>
<th>#Revisions</th>
<th>kLOC</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>Oct-09 to Nov-17</td>
<td>5382</td>
<td>14027</td>
<td>200</td>
<td>Java</td>
</tr>
<tr>
<td>egit</td>
<td>Sep-09 to Nov-17</td>
<td>5336</td>
<td>13211</td>
<td>157</td>
<td>Java</td>
</tr>
<tr>
<td>linuxtools</td>
<td>Jun-12 to Nov-17</td>
<td>5105</td>
<td>15536</td>
<td>279</td>
<td>Java</td>
</tr>
<tr>
<td>platform-ui</td>
<td>Feb-13 to Nov-17</td>
<td>4756</td>
<td>14115</td>
<td>637</td>
<td>Java</td>
</tr>
<tr>
<td>ns_server</td>
<td>Apr-10 to Nov-17</td>
<td>11346</td>
<td>34317</td>
<td>255</td>
<td>JavaScript</td>
</tr>
<tr>
<td>testrunner</td>
<td>Oct-10 to Apr-16</td>
<td>7335</td>
<td>17330</td>
<td>117</td>
<td>Python</td>
</tr>
<tr>
<td>cp-engine</td>
<td>Feb-13 to Nov-17</td>
<td>6475</td>
<td>22885</td>
<td>68</td>
<td>C++</td>
</tr>
<tr>
<td>indexing</td>
<td>Mar-14 to Nov-17</td>
<td>3240</td>
<td>8316</td>
<td>107</td>
<td>Go</td>
</tr>
</tbody>
</table>

Based on the review and revision numbers. Such versions are accessible through the commit ids provided in the projects’ CSV file, as discussed above.

We store the discussion files for each revision in the discussion directory. This directory follows a tree structure, organised by review number, in which the discussion files for each revision are contained in the directory of its respective review. A discussion file presents reviewing data in the following order: first, the description of the revision is presented, which denotes the commit message of the revision. Such a message includes the revision’s change-id and Author. The comments that were made during review by other developers are presented next. In the discussion file we include the author of the comment and the respective message.

3 At the time of writing, papyrus is the project with most code reviews in Eclipse. However, it was not the case when we started mining.
4.1 Research Directions

As previously mentioned, there has been a good number of empirical studies that explored Gerrit data for different purposes [4, 7, 11–14, 19]. Hence, the data provided by CROP can now be used for extension, replication and validation of existing studies in code review, such as the ones mentioned above, as well as the formulation of new studies.

We envision the data provided by CROP being leveraged in a series of studies that span not only code review, but also other areas of software engineering research. By linking code reviews to complete versions of the code base, we can now fully assess the impact of code review in the context of the code about which the reviews are written. Researchers may want to evaluate how code review influences building and testing, for example. Since CROP provides data for all revisions within a review, one might evaluate how the quality of a patch evolves from when it is first submitted to when it has finally been merged. We also provide source code for abandoned reviews, which enables the evaluation of, for example, whether the rejection of code reviews is somehow correlated to quality indicators in the submitted source code. By profiling developers’ patterns and behaviour for a certain system, researchers may assess the effect of code reviews on the knowledge transferred between developers as the system evolves.

During code review, developers are constantly providing reasoning and rationale for the changes they make in the system, both when they submit code for review and when they inspect code from their peers. Thus, we see code review data as a scientifically valuable source of knowledge regarding motivation for and explanation of software changes. This knowledge can be leveraged to answer questions that previously required interactions with developers, such as interviews and surveys. One might use code review data to assess how developers react to the introduction and removal of code smells, for example. Similarly, one may investigate how developers deal with code duplication. By analysing different systems, one may be able to study how different teams reason about and discuss their maintenance activities, such as refactoring and bug fixing.

CROP is an ongoing research project, where we will periodically update the code review data to reflect the evolution of the systems in the dataset. In addition, we will be constantly mining and including reviewing data for other open source systems. Finally, CROP’s code base is open4 for contributions.

REFERENCES


4https://github.com/crop-repo/crop

source code that we archived and which would otherwise no longer be available in the systems’ original repositories. By exploring the data provided in CROP, researchers can now study the effects of code review on the code base. Moreover, code review generates valuable and detailed information about software changes that can be leveraged in the development of other research areas in software engineering, such as code smells, code cloning and refactoring.