Chapter 2

Data Availability and Current Code Velocity Trends

Abstract

To determine the directional trend of code velocity, we need access to large amounts of data that quantifies several multi-year open-source software projects. Based on our existing domain knowledge, the only public code collaboration tool that collects detailed code review data is Phabricator. This chapter (a) describes the new Phabricator code review dataset that we will use for our subsequent studies, (b) introduces a novel taxonomy of code review mining smells, and (c) investigates how the duration of various code review periods changes over a project’s lifetime. To understand the trend in code velocity, we study four open-source software projects: Blender, FreeBSD, LLVM, and Mozilla. We mine and analyze the characteristics of 283,235 code reviews that cover, on average, seven years’ worth of development.

2.1 On Usage of Phabricator Code Reviews

2.1.1 Introduction

A variety of code review datasets are published. Some of the most well-known include Code Review Open Platform (CROP) [Paixao, Krinke, Han, & Harman, 2018], Gerrit code review dataset [Yang, Kula, Yoshida, & Iida, 2016], and GHTorrent [Gousios, 2013]. Several popular open-source software projects (e.g.,
FreeBSD, LLVM, Mozilla) use a code collaboration tool called Phabricator [Phabricity, 2021a] to conduct their code reviews. We have not found any published code review datasets for Phabricator. The search of existing literature about mining popular code collaboration tools reveals a study documenting the mining of Gerrit data for Android [Mukadam, Bird, & Rigby, 2013] and GitHub [Gousios, 2013]. We can locate only one thesis about mining projects using Phabricator [Cotet, 2019]. This thesis describes the development of a data mining tool called Phabry [Cotet, 2021]. Phabry, however, cannot be used to collect code changes associated with a code review.

The absence of a readily accessible dataset of code changes for Phabricator projects has deprived code review researchers of a rich information source. The benefit of Phabricator is the ability to formally distinguish between different events taking place during the code review. Each event and action taken during the life cycle of a code review is associated with an author’s identity and an event’s timestamp. Researchers can track when a code review was accepted, abandoned, taken over by someone else, when a reviewer resigned, when some attributes (e.g., title) were updated, etc.

Table 2.1: Different events during Phabricator code review.

<table>
<thead>
<tr>
<th>abandon</th>
<th>create</th>
<th>reopen</th>
<th>subscribers</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>draft</td>
<td>request-changes</td>
<td>summary</td>
</tr>
<tr>
<td>author</td>
<td>inline</td>
<td>request-review</td>
<td>testPlan</td>
</tr>
<tr>
<td>close</td>
<td>plan-changes</td>
<td>resign</td>
<td>title</td>
</tr>
<tr>
<td>commandeer</td>
<td>projects</td>
<td>reviewers</td>
<td>update</td>
</tr>
<tr>
<td>comment</td>
<td>reclaim</td>
<td>status</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 lists all possible events we noted during the code review life cycle of popular Phabricator projects. We are not aware of any other code review system that tracks events with this level of granularity. GitHub introduced the more formalized code review process, including functionality for actions such as formal acceptance of code changes, only in 2016 [GitHub, 2016]. By not utilizing publicly available Phabricator data, researchers miss out on potentially valuable insights and opportunities to study influential and popular software projects with a multi-year development history.

Without a pre-existing accessible dataset, we set out to acquire the Phabricator data ourselves and convert the data to a format suitable for further analysis. Based on our experience, reliably mining data associated with hundreds of
thousands of code reviews, even with a pre-existing tool, is an involved and
time-consuming process requiring a nontrivial amount of manual labor. We
describe the challenges encountered and our solutions in Section 2.1.3.

The primary motivation behind our work is to publish a dataset that (a) does not require extra mining effort, (b) includes data about code changes in the code reviews (files changed; lines of code added, deleted, or updated), and (c) can be imported into a relational database system such as MySQL in addition to being published in a plain JSON format.

2.1.2 History and overview

Phabricator was initially developed as an internal code review tool for Face-
book in 2011 [Tsotsis, 2011]. As of this chapter (November 2021), it is still the
de facto code review environment for Facebook and is internally under active
development. The public version of Phabricator is developed by a company
called Phacility and distributed as open-source software [Phacility, 2021a].

When compared to other well-known code review environments, such as Ger-
rit or GitHub, Phabricator introduces some new code review related terminology. For example, the proposed code modifications in Gerrit are referred to as change (same as pull requests in the context of GitHub). A code review iteration in Gerrit is a version of the change and is called patch set. In Phabricator both the initial set of code modifications and its subsequent versions are called differential revision, which gets shortened to a diff. Committing and merging the accepted changes to the target branch is called submitting in Gerrit and landing in Phabricator.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type of software</th>
<th>Year of first diff</th>
<th>Total reviews</th>
<th>Accessible reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blender</td>
<td>Graphics</td>
<td>2013</td>
<td>13,151</td>
<td>13,097 (99.59%)</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>OS</td>
<td>2013</td>
<td>32,884</td>
<td>32,725 (99.52%)</td>
</tr>
<tr>
<td>KDE</td>
<td>Desktop</td>
<td>2015</td>
<td>29,953</td>
<td>29,874 (99.73%)</td>
</tr>
<tr>
<td>LLVM</td>
<td>Compiler</td>
<td>2012</td>
<td>113,372</td>
<td>112,892 (99.58%)</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Browser</td>
<td>2017</td>
<td>130,567</td>
<td>128,888 (98.71%)</td>
</tr>
</tbody>
</table>

A wide range of Phabricator projects are publicly accessible. Table 2.2 lists the projects published by our dataset. We describe the type of project, when the first differential revision was published, the amount of available code review
data as of November 2021, and the percentage of code reviews that are publicly accessible. The median age of a project is 8 years and the median number of unique contributors per project is 1,504. Out of 317,476 code reviews, only 258 (0.08%) do not have any associated data quantifying the code changes. The lack of data is caused by changes consisting of binary files or containing only renaming of files.

### 2.1.3 Mining data

#### Authentication and data access

The Phabricator user community maintains a list of organizations and projects that utilize the tool [Phabricator, 2021e]. We find that resource, in addition to our knowledge from industry experience, to be the best available reference related to Phabricator’s usage. Interaction with Phabricator is conducted via Conduit API [Phabricator, 2021b]. Conduit API is a remote procedure call protocol where requests and responses are encoded in JSON (JSON-RPC). To mine Phabricator data, the client needs to have a Conduit API token for authentication purposes. Acquiring the token requires creating a user account for each Phabricator instance to be mined. Account creation can either require a manual approval from a member of the development team (FreeBSD), possession of the GitHub account (Mozilla, LLVM), or just filling out the required registration data (Blender, KDE).

The official API documentation contains only a limited number of examples about its usage. We find that practical experimentation with curl [cURL project, 2021] or API console is the most efficient way to gain knowledge [Phabricator, 2021g]. To mine the data, one can develop their own tool(s) (something authors of this chapter initially did) or utilize existing API wrappers for different programming languages [Phabricator, 2021a]. For our past code review related studies, we have utilized a version of Phabry with minor modifications to facilitate the debugging and adjustments necessary to mine different Phabricator instances [Cotet, 2021]. We find that the thesis describing Phabry’s development is a detailed and valuable reference about how to interact with Conduit API [Cotet, 2019].

#### Parsing and interpretation

Data retrieved via Conduit API is returned in JSON format. Our initial instinct was to follow the approach taken in both Gerrit and GHTorrent datasets and import the data into a database such as MySQL [Gousios, 2013; Yang et al., 2016]. Though the output from Conduit API is not documented, building a relational...
normalized database schema was a straightforward process. The downside of exposing the dataset as a database is the cost associated with maintaining the database instance, importing data, deciding what fields to index, etc. For our studies, we both parse and extract data from JSON directly and use SQL to mainly gather descriptive statistics. That approach proves to be performant even with dataset sizes between 2–3 GBs and up to 135,000 files. The dataset we expose contains both raw JSON files and MySQL database containing the same information.

**Associating differential revisions with code**

Each differential revision can evolve through multiple versions. Code changes between each version can differ. To understand the full evolution of the code review it is necessary to keep track of how the code review evolved over the time. However, most code review related studies limit themselves to only the initial or the final version of code changes. In addition, our intent is not to duplicate the data stored in the source control system. For our dataset, we keep track of number of files changed and lines added, deleted, and updated for the final version of the differential revision. We use `diffstat` to calculate the code churn statistics from the raw diff output [Dickey, 2021].

There are multiple options for mapping the final code changes to differential revisions. The intuitive approach is to inspect the commit history of a source control system and match the commit content with a differential revision. Listing 2.1 displays a randomly picked FreeBSD commit using a Phabricator code review process.

**Listing 2.1: Anonymized FreeBSD commit description.**

```plaintext
commit mrmauqlsdpdymqchdtmnmadcmakrzeqjei1
Author: John Doe <john.doe@FreeBSD.org>
AuthorDate: 2971410770
Commit: John Doe <john.doe@FreeBSD.org>
CommitDate: 2971410770
  foo: fix a memory corruption in bar.
Differential Revision: https://reviews.freebsd.org/D12345678
```

Based on our analysis, the presence of the string associating a commit with the specific differential revision is *optional* and depends on the project. In addition, we observe typographic errors in the URLs referencing differential revisions and using different notations when referring to a code review. Therefore, we cannot reliably use the data from commit descriptions to determine what differential
A single commit can tag multiple differential revisions and a single differential revision can be referenced from multiple commits. Fetching the data about code changes directly from Phabricator results in a correct representation of final code changes.

**Challenges**

**Networking** The server hosting Phabricator may apply rate limiting to the number of requests a Conduit API client can issue or the number of network connections the client can make overall. We find that it is necessary to have a retry mechanism in place to mitigate the presence of intermittent errors such as server returning a variety of HTTP error codes, connections timing out, etc. Depending on the specifics of a Phabricator instance, the server may also require a HTTP GET request for one project and PUT request for another (e.g., Blender).

**Permissions** During our data mining process we found that there is a subset of differential revisions accessible only to authenticated users, i.e., they cannot be directly downloaded via curl [cURL project, 2021] without providing the required Conduit API token. We utilize the subsequent usage of getting the metadata about differential revision from differential.query [Phabricator, 2021d] and using it to fetch the raw content by calling differential.getrawdiff [Phabricator, 2021c].

However, there were some revisions which even an authenticated user could not access. Those differential revisions were a minor part of the overall dataset, accounting for a median of 0.42% of differential revisions per Phabricator instance.

**API evolution** Phabricator is distributed as open-source software and each project is free to make any changes needed for their purposes. Depending on the Phabricator instance, the type of data returned by Conduit API may be different. Differences may manifest in the data fields present, action types that can be performed on a differential revision, and if certain fields are optional or mandatory. In some cases, even the data type of the field varies between different Phabricator instances.
2.1.4 Database schema

Design decisions and data representation

One of the initial design decisions we faced was a choice of exposing the data in the database as close to its original representation in JSON versus using a third normal form [Garcia-Molina et al., 2009]. Third normal form is used to reduce data duplication, amount of storage required, and increase the performance of database queries. For simplicity, we chose to match the JSON structure as much as possible unless normalization was needed to represent entries of variable count.

The full relational database schema is presented in Appendix A. Each table has a primary key called \textit{Id}. The foreign key columns referencing parent tables are prefixed with \textit{FK} and end with the name of a referring table. In modeling the data, we chose to follow the Phabry output directory structure and the Phabricator design concepts. Two essential tables are \textit{revisions} and \textit{transactions}. Transactions in the context of Phabricator are the history of edits associated with each revision [Phabricator, 2021h]. Each revision belongs to a single Phabricator instance stored in the \textit{instances} table. One revision can have many transactions associated with it. Each transaction belongs to only a single revision. Each revision can have many reviewers and subscribers related to it. A revision can belong to many projects. Each transaction can be associated with many comments, inline comments, a set of changed fields, or many commits. The timestamps (\textit{dateCreated} and \textit{dateModified}) are in Unix time (number of seconds since the Epoch) [IEEE and The Open Group, 2021] and represented as integers.

```
Listing 2.2: Anonymized FreeBSD revision in JSON format.
{
  "id": 1234567890,
  "type": "DREV",
  "phid": "PHID-DREV-viqvtimavobvsqvgbugp",
  "fields": {
    "title": "Title description",
    "uri": "https://reviews.freebsd.org/D1234567890",
    "authorPHID": "PHID-USER-gnuefszwyfdzrescjih",
    "status": {
      "value": "published",
      "name": "Closed",
      "closed": true,
      "color.ansi": "cyan"
    },
    "repositoryPHID": "PHID-REPO-tucbfqmbgohbfcczvfg",
    "diffPHID": "PHID-DIFF-hmkchkgiochcmfmg",
    "summary": "Summary of the code changes."
  };
```

2.1. On Usage of Phabricator Code Reviews

2.1.4 Database schema

Design decisions and data representation

One of the initial design decisions we faced was a choice of exposing the data in the database as close to its original representation in JSON versus using a third normal form [Garcia-Molina et al., 2009]. Third normal form is used to reduce data duplication, amount of storage required, and increase the performance of database queries. For simplicity, we chose to match the JSON structure as much as possible unless normalization was needed to represent entries of variable count.

The full relational database schema is presented in Appendix A. Each table has a primary key called \textit{Id}. The foreign key columns referencing parent tables are prefixed with \textit{FK} and end with the name of a referring table. In modeling the data, we chose to follow the Phabry output directory structure and the Phabricator design concepts. Two essential tables are \textit{revisions} and \textit{transactions}. Transactions in the context of Phabricator are the history of edits associated with each revision [Phabricator, 2021h]. Each revision belongs to a single Phabricator instance stored in the \textit{instances} table. One revision can have many transactions associated with it. Each transaction belongs to only a single revision. Each revision can have many reviewers and subscribers related to it. A revision can belong to many projects. Each transaction can be associated with many comments, inline comments, a set of changed fields, or many commits. The timestamps (\textit{dateCreated} and \textit{dateModified}) are in Unix time (number of seconds since the Epoch) [IEEE and The Open Group, 2021] and represented as integers.

Listing 2.2: Anonymized FreeBSD revision in JSON format.
```
For example, sample JSON content in Listing 2.2 represents a subset of a record in the revisions table.

2.2 Code Review Mining Smells

2.2.1 Introduction

Mining data related to code reviews can help researchers to discover new insights about various issues associated with the software development process. The correct interpretation of the mined code review data needs precision and formality. In the past, researchers have studied pitfalls associated with mining version control systems such as Git [Bird et al., 2009], code collaboration tools such as GitHub [Kalliamvakou et al., 2014], and general aspects of ethics and privacy in data mining [Gold & Krinke, 2021; Gonzalez-Barahona, 2020]. Based on our industry experience, interpretation and decisions related to what constitutes a valid code review dataset influences the outcome of the findings. We are unaware of existing studies that enumerate various open issues related to nuances of mining code reviews.

Decision points that a researcher encounters can include (a) formally defining the code review states, such as when it is accepted and ready to be committed, (b) choice of algorithms and tools to calculate quantitative metrics, such as the size of the code review, and (c) decisions about how to define outlier code reviews. We find only a few papers that include formal definitions or detailed discussions about these issues in the “Methodology” or “Threats to validity” sections. For the papers that do not have a replication package, determining what was measured and how various terms are defined is up to interpretation. That makes the replication of results or comparison with the existing findings error-prone.

This chapter categorizes various smells associated with mining code reviews, provides potential recommendations with mitigation techniques, and lists open issues for which authors do not have a definitive solution.
2.2. Code Review Mining Smells

2.2.2 Background and motivation

The term code smell is traditionally associated with issues related to programming [Tufano et al., 2015]. The canonical definition of a code smell “is a surface indication that usually corresponds to a deeper problem in the system” [Fowler, 2006a]. Various code smells exist, and researchers have studied this topic in-depth [Sharma & Spinellis, 2018]. Based on our research on code reviews, we encounter similar smells when interpreting the research papers or using the existing code review datasets. We define the data mining smell as an issue related to data mining, the impact of which is not discussed or a critical decision the authors do not explain.

Mining code review data is a costly and cumbersome process. Researchers need to overcome the rate-limiting restrictions of the queried system and handle large volumes of data [Gousios & Spinellis, 2012]. Several datasets about code reviews that help researchers to be more efficient have been published [Gousios, 2013; Paixao et al., 2018; Yang et al., 2016]. These datasets cover code collaboration tools such as Gerrit or GitHub. Researchers typically explicitly trust the quantitative metrics from a dataset. Acting otherwise would defeat the purpose of using a pre-mined dataset.

The research related to code reviews is mainly quantitative and explores the topics such as duration [Yu, Wang, Filkov, Devanbu, & Vasilescu, 2015], frequency [Izquierdo-Cortazar et al., 2017], sentiment [Ahmed, Bosu, Iqbal, & Rahimi, 2017], and size [Söderberg et al., 2022]. The general description of how the code review data was gathered, filtered, interpreted, or calculated only answers some questions. Even when consulting with other practitioners and researchers in academia and industry, we have yet to reach a clear consensus about how to interpret a specific term or calculate a metric.

To improve the correctness of the data mining process, we composed a list of issues that indicate problems related to code review data mining that we have encountered in the past. Most of the smells we categorize in Section 2.2.4 relate to the need for formality when defining terms or deciding how to interpret data. We propose solutions to most of the data mining smells in Section 2.2.5. We aim to trigger a discussion in the mining software repositories community about the issues that need a solution.
2.2.3 Methodology

The list of the code review mining smells is based on our experience when interpreting the data from existing code review datasets [Gousios, 2013; Paixao et al., 2018; Yang et al., 2016], developing a mining infrastructure for GitHub and Phabricator, and trying to replicate findings from other research papers. Each time we encountered an unclear decision point, ambiguous definition, or any issue that triggered an extended discussion, we added a specific problem to our list of items to check for and validate. We consider the taxonomy of issues to be partial. Our classification serves as a starting point for a broader discussion.

2.2.4 Taxonomy of issues

In Table 2.3, we display different categories of code review mining smells. That classification represents the common problems we have encountered during our research into code reviews. In the following subsections, we describe each smell in more detail and offer a potential course of action.

Table 2.3: A taxonomy of potential code review mining smells.

<table>
<thead>
<tr>
<th>Category</th>
<th>List of smells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptance</td>
<td>Self-acceptance, Never accepted, Re-opened, Multiple accepts</td>
</tr>
<tr>
<td>Quantitative</td>
<td>Timestamp inconsistency, Timestamp source, Period duration, Size calculation</td>
</tr>
<tr>
<td>Content</td>
<td>Language differences, Binary content, Merges and dependencies, M : N for commits and reviews</td>
</tr>
<tr>
<td>Context</td>
<td>Source versus mirror repository, Branch selection, Inaccessibility, Review buddies, Bots and humans</td>
</tr>
</tbody>
</table>

Acceptance of code reviews

Self-accepted code reviews A lack of responses to the code review can cause frustration or delay the completion of a feature. To make progress and be able to commit the code changes, an author will accept their changes to “satisfy the process.” In case of negative consequences, the typical justification we have witnessed is “it is better to ask forgiveness than permission.”

The Modern Code Review process [Bacchelli & Bird, 2013; Sadowski, Söderberg, et al., 2018] assumes that someone other than the author of the code changes reviews these changes. For example, a project can require that all the code reviews are formally conducted via a code collaboration tool and signed off. We
observe that a nontrivial percentage of code reviews are self-accepted. Between 2012 and 2022, there were 152,541 code reviews in the LLVM project. Of them, for 3,506 (2.3%) code reviews, the author of changes accepted their code changes. We think self-accepted code reviews do not constitute a good code review because the review did not take place.

Reviews that were never accepted The code review guidelines for open-source software projects such as FreeBSD [The FreeBSD Documentation Project, 2022a], LLVM [LLVM Foundation, 2023a], and Linux [The Linux Foundation, 2022] set an expectation that code review can take a noticeable amount of time. The accepted recommended interval for “pinging” the reviewers in open-source software is approximately a week. When nobody responds to a code review, an author faces a dilemma. They can either wait indefinitely, remind the potential reviewers periodically to inspect the changes or proceed forward. Between 2012 and 2022, there were 37,815 code reviews in the FreeBSD project. Of them, authors committed 2,855 (7.5%) without any acceptance. Similarly to the self-accepted code reviews, we think these reviews need to be removed from the analysis.

Another interesting subcategory we have encountered is the code reviews that were accepted after the changes were already committed. The existing research [Baum, Kortum, Schneider, Brack, & Schauder, 2016] that studies review-then-commit and commit-then-review models shows “that there are many situations with no practically relevant difference between both choices.” However, to follow the Modern Code Review guidelines, we think that these code reviews should be also excluded from the analysis.

Re-opened code reviews A standard Modern Code Review workflow takes place when the code changes are reviewed, accepted, and committed. Sometime later, either a problem with code changes was discovered, or someone not part of the initial discussion, decided to resurrect the debate about the correctness of changes. An author or someone else can re-open the original code review or pull request and restart the discussion. The newly resurrected code review then goes through a similar set of steps as the original one.

For the data analysis, this situation presents a challenge. Should we ignore the subsequent iterations of this code review, or should we treat it as multiple separate code reviews? Most likely, the individuals involved in the discussion are already familiar with the changes and context. However, the passage of time also justifies treating the new iteration as a different code review.
Multiple accepts required Some projects, such as OpenStack [OpenDev, 2021], require multiple reviewers to sign off on changes before they are considered to be properly accepted. Code collaboration tools, such as Gerrit [Google, 2021a], enable administrators to specify the acceptance policy. Researchers need to ensure that they will not apply an obvious algorithm that looks for the first acceptance. One complication of this scenario is when projects change their acceptance policy midway. Another corner case is related to multiple accepts when a secondary reviewer accepts changes before the primary reviewer, such as an area owner. In those cases, an acceptance from the secondary reviewer does not count and needs to be ignored.

Quantitative metrics

Inconsistent timestamps Researchers can, in parallel, use data from the version control system and code collaboration tool. That helps researchers to either gather missing information (e.g., the code collaboration tool does not indicate when the changes were committed) or supplement the data (e.g., commits that have code review information in the description but were not formally reviewed using the tool). We have experienced inconsistent timestamps in the version control system and code collaboration tools. For example, Git commits have both the AuthorDate and CommitDate fields. The common assumption is that changes are committed after they have been authored. However, we observe a few commits where this is false.\(^1\) Similarly, we notice commits where the values of both fields are set to 0.\(^2\) We consider these cases invalid data and recommend that they be removed from the dataset.

In the code reviews, we can also establish a sequence of events that are supposed to happen. For example, the acceptance timestamp should be later than the code review creation time, or the code review should be closed after it was accepted. We recommend that researchers filter out the code reviews that do not meet these requirements.

The source of the timestamp When a code review is closed or marked as merged, one option for calculating its lifetime is to use the timestamp from the code collaboration tool. Another option is to use the timestamp from the version control system that documents when the changes were committed to the destination branch. The problem manifests when those timestamps have a significant

\(^1\)https://github.com/llvm/llvm-project/commit/57f408861
\(^2\)https://github.com/mozilla/gecko-dev/commit/220fcaee
2.2. Code Review Mining Smells

 difference. For example, 15 seconds could be an acceptable discrepancy between the two systems. However, what about 3–4 hours? Which timestamp should the researchers use? We recommend that researchers be explicit about the context that they measure.

Duration of different code review periods  The Modern Code Review process assumes that reviewers inspect the code. However, there are code reviews that are immediately accepted or signed off only after a very short period. For example, if the size of the code review is 1,500 source lines of code (SLOC), is the review time of 17 seconds valid? Should we filter out this code review because we assume that the review did not take place? The code review could have happened offline, and engineers used the code collaboration tool to document this. Should there be a minimal review time for good code reviews?

What is the size of a code review  Code collaboration tools, such as GitHub, express the size of the code reviews as the sum of added and deleted lines. The Myers algorithm is a common way to diff two text versions [Myers, 1986]. Git uses the Myers algorithm by default [Nugroho, Hata, & Matsumoto, 2019]. However, Git also supports Minimal, Patience, and Histogram diffing algorithms. A study finds “different values in 1.7% to 8.2% commits based on the different diff algorithms” and recommends the usage of the Histogram algorithm for more precise results [Nugroho et al., 2019]. Should the calculations indicate what algorithm was used, or should the default diffing algorithm be changed?

Another dimension is the need for granularity when using only added and deleted lines. There is a third category of changes—updated or modified lines. Tools, such as diffstat [Dickey, 2021], can calculate all three categories. Most studies related to code reviews use only added and deleted lines.

The content of code reviews

Are all the source lines of code the same  In practice, code reviews are published for various content. Most code reviews contain text written in programming languages such as C, Erlang, Python, or Rust. The other content can include (a) plain text, such as documentation or installation instructions, (b) markup language that is not necessarily a programming language, such as HTML, JSON, or XML, and (c) descriptive logic, such as the rules in a Makefile. Should we treat this content the same? It is reasonable to assume that 100 lines of device driver
code that is supposed to run in kernel mode will get more scrutiny than 100 lines of documentation. Should we assign weights to different content categories?

**No text to review** Some code reviews can contain only binary content such as audio, images, or video files. Using lines to classify this content is inapplicable. However, it is reasonable to assume that someone is still reviewing an image or listening to audio. How do we quantify that work and compare it to traditional code reviews?

**Merges and dependencies** During software development, merging changes from one branch to another is common. For example, a developer may merge three months’ worth of work from a milestone-specific branch to a main branch. Such merges can contain thousands to hundreds of thousands of SLOC. While officially, those merges are submitted as code reviews, de facto, no code review takes place. Should we exclude code reviews that are bigger than a specific size?

Similarly, there are cases when a product is dependent on an open-source software library in the form of source code. Whenever a new version is released, then commit that updates the dependency can contain tens or hundreds of thousands of SLOC. It is reasonable to assume that no actual code review takes place to review all the dependencies. Should these types of updates be removed from the analysis?

**M : N relationship between commits and code reviews** One of the most complicated cases to resolve automatically is when there is an M : N relationship between commits and code reviews. While it is uncommon, a single code review can be split between multiple commits. Similarly, one commit can refer to multiple code reviews. For example, a commit can fix one defect and part of the problem another code review is trying to solve. In our experience, even after conducting manual analysis, it is hard to determine how to divide the number of code changes between code reviews or when to consider the code review to be complete.

**Context**

**Use the GitHub mirror or original** Software projects can use version control systems such as Mercurial, Perforce, or Subversion. It is common for popular projects to have a read-only GitHub mirror of their source code. For example, projects like OpenBSD use CVS as the version control system and provide a
2.2. Code Review Mining Smells

public GitHub conversion mirror. If researchers mine multiple projects, the path of the least resistance is to use the read-only GitHub mirror. That decision causes the usage of only one toolset to mine the source code history. Researchers assume the conversion from one version control system is lossless and maintains the same data. The authors should document this assumption in the paper.

**Does the branch matter** Using different branches is standard in software development. A product may have (a) a main branch for the daily development efforts, (b) a release branch where the next release is prepared, and (c) a branch where experimental or prototype work is done. For example, Mozilla has a well-defined branch structure and release management process that governs how the code moves between different branches. Should the code reviews from all these branches be treated equally? It is reasonable to assume that if we consider the scrutiny for the main branch to be at a certain level, then code that goes into an experimental branch may receive less scrutiny. Similarly, the code changes to the release branch, where only fixes that block the release are accepted, will receive more reviewer attention. Should we treat the changes in those branches somehow differently?

**Inaccessible code reviews** Some projects may restrict access to a subset of code reviews for various reasons. For example, a team may discuss a fix to a potential zero-day vulnerability that has yet to be public. The content for those code reviews could potentially never be made accessible to anyone outside the core team. As a result, those code reviews are not visible to researchers mining the code review data.

Similarly, a code collaboration tool can return no data or an internal error (e.g., a 5xx HTTP status code) for a certain amount of code reviews. Various software defects, such as data inconsistency or service unavailability, typically cause this behavior. There is little external researchers can do in this case.

**Review buddies and stamping of code reviews** Organizations can establish rigid rules about following the code review process. For example, Meta requires each diff to be reviewed without exceptions [Riggs, 2022]. Based on our industry experience, we observe that engineers use simple workarounds to avoid the perceived overhead of the code review process. If Alice and Bob are colleagues,

---

3 [https://github.com/openbsd](https://github.com/openbsd)
they can informally agree to “stamp” (i.e., immediately accept without a review) each other’s changes. We have anecdotally witnessed this behavior in multiple organizations between two or more engineers who have explicit trust in each other’s ability to produce quality code. Those engineers were willing to accept the consequences of occasional defects if it helped to make code reviews faster. The “stamping” pattern is generally identified by a very short time-to-accept between a fixed number of individuals. Should we consider these code reviews to be valid?

**Bots versus humans**  To measure various code review periods, such as time-to-first-response, researchers can look at when there is the first evidence of activity by someone other than an author of changes. Modern code collaboration tools involve various bots. Those bots can post updates to code reviews, such as the indication of the test validation, notify the author that their commit description does not match project-wide standards, or include additional reviewers based on the context. Determining if an actor is a bot or a human is a complex problem [Golzadeh, Decan, Constantinou, & Mens, 2021]. Assuming that we can do that, should we remove all the activity caused by bots? Should we treat bot activity as a “real response” to the code reviews.

### 2.2.5 Recommendations

This section enumerates ten recommendations for researchers that mine code review data. The context for these recommendations is to investigate the aspects of the code review process that relate to code velocity. An exception to avoiding these smells is when researchers intend to study specific phenomena such as re-opened code reviews.

1. Remove the completed code reviews that were self-accepted, never accepted, or where the acceptance was recorded after the code changes were committed.

2. Treat re-opened code reviews that went through multiple complete iterations as different and independent code reviews.

3. In cases of multiple accepts, be familiar with the project’s policy and change the data analysis logic accordingly.
4. Remove all the commits and code reviews where timestamps are inconsistent or invalid (e.g., the commit happens before the code review was submitted for review).

5. Remove all the code reviews that do not contain anything that can be measured in SLOC.

6. Filter out all the activities that humans do not cause.

7. Use added, deleted, and modified (updated) SLOC to classify code churn.

8. Document the number and percentage of inaccessible code reviews.

9. Remove the prominent non-code reviews, such as merges or updates to the source code of dependencies.

10. Mine the data from the original version control system and avoid using conversion mirrors.

2.2.6 Discussion

A decision about the appropriate level of detail and reasoning behind the choices related to data mining is a balancing act. Researchers have limited space to discuss the filtering process and formal definitions of various metrics. A reasonable approach is to require that the replication package contains at least documentation about how most potential code review data mining smells are handled.

Standardization of definitions is another possible avenue to explore. Most projects use a few popular code collaboration tools and version control systems. Defining the standard terms and states for each environment and tool is necessary. For example, periods such as the lifetime of a code review or algorithm and tools used to measure its size.

2.2.7 Threats to validity

Our study is subject to specific categories of threats [Shull et al., 2008]. One of the threats to external validity is selection bias. The data mining smells we determined come from analyzing only a small subset of major open-source software projects. Those projects use a limited number of code collaboration tools (Gerrit, GitHub, Phabricator). It is feasible that some smells are no longer a concern in a more regulated and formal environment. Certain commercial software types that
follow specific ISO standards belong to that category [ISO Technical Committee, 2017]. For example, in a project where the policy states that a single commit accompanies each code review, and each commit must result from a code review.

We draw inferences from a sample of open-source software projects and focus on specific aspects of code reviews for conclusion validity. Researchers with a different focus may have classified the smells differently.

2.3 On Temporal Aspects of Code Velocity

2.3.1 Introduction

One critical goal in the software industry is to develop, review, integrate, and deploy code changes fast. The software industry focuses on increasing the code velocity [L. Chen et al., 2022; Maddila et al., 2022; Riggs, 2022] using different tools or process enhancements. Various adaptations of Continuous Integration (CI) and Continuous Deployment (CD) [Fowler, 2006b] have become default practices for most of the current projects in both industry and open-source software communities. Similarly, the lightweight Modern Code Review [Bird et al., 2015; Sadowski, Söderberg, et al., 2018] is now a de facto standard process to conduct code reviews. For existing non-agile projects, the initial switch from methodologies such as the waterfall model to CI/CD or mailing lists to contemporary code collaboration tools can have an immediate and noticeable impact on increasing code velocity [Jha, Vilardell, & Narayan, 2016; Kaur, Khurana, & Manisha, 2021]. The longevity of these improvements to code velocity has yet to be thoroughly investigated.

We focus on studying the direction of a change in the duration of various code review periods as the software projects evolve. We intend to determine how the speed of code reviews changes over time. We mine code review data from four different open-source software projects: Blender [Blender Foundation, 2023], FreeBSD [FreeBSD Foundation, 2023], LLVM [LLVM Foundation, 2023b], and Mozilla [Mozilla Foundation, 2023b]. We analyze 283,235 code reviews and evaluate how different code review periods (time-to-first-response, time-to-accept, and time-to-merge) trend over time. Our study suggests that the speed of code reviews remains the same as the projects evolve.
2.3.2 Background and motivation

Different interpretations and scopes for code velocity exist. A general definition is “the time between making a code change and shipping the change to customers” [Microsoft Research, 2019]. In this chapter, we focus on a more quantifiable metric related to the duration of code reviews. We use time-to-merge as a proxy metric to quantify how fast code changes propagate. The time-to-merge covers a period from when an engineer publishes a set of code changes that are ready for code review till these changes are merged to the target branch [Izquierdo-Cortazar et al., 2017].

There are two prevailing and contradictory theories within industry about the direction of code review velocity over time. The first hypothesis states that due to an increase in the size and complexity of a system (Lehman’s second law “Increasing Complexity” [M. Lehman, 1980]), e.g., increase in the complexity of communication due to a bigger team, code velocity decreases. A second hypothesis argues that code velocity increases over time. The increase in code velocity is because engineers become more familiar with the code base, interpersonal communication becomes more efficient, and the tooling infrastructure improves.

Another of the Lehman’s laws of software evolution is called “Continuing Growth” [M. Lehman, 1980]. Lehman’s sixth law states that the “[f]unctional content of a program must be continually increased to maintain user satisfaction over its lifetime” [M. M. Lehman, 1991]. As a result of additional functionality, it is reasonable to assume that the size of the code base increases. The size of the code base is typically measured in source lines of code (SLOC). Based on our industry experience, we also observe that in conjunction with the new demands on a project, the size of the development teams tends to increase rather than decrease. According to Brooks’ law, “[a]dding manpower to a late software project makes it late” [Brooks, 1995]. While the software project does not necessarily have to be late while developing new features, it is unknown how increased code base and team sizes impact code velocity. To investigate this subject further, we formulate the following research question:

**RQ:** How does a project’s code velocity trend over time? Does the code velocity increase, decrease, or stay neutral?
2.3.3 Methodology

Choice of data

Section 2.3.2 states that our primary code velocity metric of interest is time-to-merge [Izquierdo-Cortazar et al., 2017]. We also know that research from both Google [Google, 2023] and Microsoft [Bird et al., 2015] finds that time-to-first-response and time-to-accept are additional code review metrics important for developers.

Existing code review datasets focus either on GitHub [Gousios, 2013] or Gerrit [Mukadam et al., 2013; Paixao et al., 2018; Yang et al., 2016]. By default, Gerrit immediately merges changes once they are accepted. That behavior means that treating time-to-accept and time-to-merge as separate events is neither valuable nor valid. GitHub added the ability to approve changes only in 2016 [GitHub, 2016]. We randomly chose 100 GitHub projects and inspected pull requests in those projects. The adoption rate and consistent use of that feature still need to be higher to yield valuable data.

One code collaboration tool that exposes a data model that formally tracks various code review periods is Phabricator [Phacility, 2021a]. Based on public information about existing Phabricator projects [Phabricator, 2021e], we mine data for four major open-source software projects with a multi-year development history. Those projects are Blender, FreeBSD, LLVM, and Mozilla. Our initial dataset contains 283,235 code reviews. We removed the code reviews where the Modern Code Review did not happen. Examples of nonconforming code reviews are the ones that do not contain any lines of code, where the author has accepted their changes, or code reviews that were committed without any acceptance. After filtering and applying consistency checks, the final dataset contains 280,456 code reviews.

Statistical analysis

As a first step, we investigate if there is a trend in code velocity. We use the Mann-Kendall test [M. Kendall, 1976; Mann, 1945] to determine if there is a monotonic upward or downward trend. The monotonic trend [Hirsch, Slack, & Smith, 1981] means that “the variable consistently increases (decreases) through time, but the trend may or may not be linear” [Pacific Northwest National Laboratory, 2022]. The null hypothesis ($H_0$) for the Mann-Kendall test is that there is no monotonic trend. The alternative hypothesis ($H_1$) is the presence of a monotonic trend. Secondly, if a statistically significant Mann-Kendall correlation is present,
then we calculate the Sen’s slope (Theil-Sen estimator) [Gilbert, 1987; Sen, 1968] to evaluate the magnitude of the trend. Sen’s slope indicates the rate of change per unit time step.

How to handle autocorrelation is a challenge for time series analysis. One popular approach is to aggregate the time series to use coarser time granularities such as monthly or yearly samples [Coen et al., 2020]. Another mitigation is to use a modified Mann-Kendall test to adjust for autocorrelation [Hamed & Rao, 1998; Yue, Pilon, Phinney, & Cavadias, 2002]. We use both techniques to reduce the chance of falsely concluding that a trend is present when it is not.

We conduct these calculations for three code review periods: time-to-first-response, time-to-accept, and time-to-merge. We look at the 30-day rolling (moving) and all-time median. The all-time median is the median of the specific metrics up to a given time. Both metrics help to evaluate the trend of a specific variable [Hyndman, 2011]. We chose the median values as opposed to the mean values because the median is more resistant to outliers. We use an $\alpha$ level of .05 for our statistical tests.

### 2.3.4 Results

Table 2.4: The $p$ values from the modified Mann-Kendall (MK) test and magnitude of change for Sen’s slope (Theil-Sen estimator). We present the 30-day moving median and all-time median for different code review periods. Statistically significant Mann-Kendall $p$ value indicates the presence of either an upward or downward monotonic trend. Sen’s slope (presented at a 95% confidence level) identifies the magnitude of the trend per unit time step. The time step is 30 days. The unit for code review periods is hours.

<table>
<thead>
<tr>
<th>Name</th>
<th>Time-to-first-response</th>
<th>Time-to-accept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30-day median</td>
<td>All-time median</td>
</tr>
<tr>
<td>Blender</td>
<td>MK</td>
<td>.231</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>$&lt; .001$</td>
<td>$0.001$</td>
</tr>
<tr>
<td>LLVM</td>
<td>$&lt; .001$</td>
<td>$0.000$</td>
</tr>
<tr>
<td>Mozilla</td>
<td>$&lt; .001$</td>
<td>$0.001$</td>
</tr>
</tbody>
</table>
Table 2.4: The $p$ values from the modified Mann-Kendall (MK) test and magnitude of change for Sen’s slope (Theil-Sen estimator) (continued). © 2023 IEEE.

<table>
<thead>
<tr>
<th>Name</th>
<th>30-day median</th>
<th>All-time median</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MK</td>
<td>Sen’s</td>
</tr>
<tr>
<td>Blender</td>
<td>.609</td>
<td>.666</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>.193</td>
<td>&lt; .05</td>
</tr>
<tr>
<td>LLVM</td>
<td>&lt; .001</td>
<td>−0.009</td>
</tr>
<tr>
<td>Mozilla</td>
<td>&lt; .01</td>
<td>−0.001</td>
</tr>
</tbody>
</table>

The results from the statistical analysis are displayed in Table 2.4. The main observations from that analysis are the following: (a) except for Blender, there is a statistically significant monotonic trend for all the projects, (b) while a statistically significant trend is present, based on its numeric values and visual representation (see Figure 2.1), it is minimal, and (c) all the 30-day median slope values are negative or zero, which suggests a *minor increase in code velocity*.

Using a 30-day rolling mean or median is standard practice. Depending on the context, 90-day (quarterly) technical indicators are also used to check for trends. We also calculate the Mann-Kendall and Sen’s slope values as an additional data point using the 90-day rolling median. The conclusions do not change as a result.

Complimentary to the analysis above, the visualization of the trend for 30-day rolling median time-to-merge is displayed in Figure 2.1. Based on visual observation, we note that for FreeBSD, LLVM, and Mozilla, most median values stay in a relatively fixed range.

For Blender, there is a noticeable cluster of results that indicate increased code velocity between 2017 and 2019. To further investigate this result, we inspected 50 random Blender code reviews between 2017 and 2019. We cannot find conclusive evidence that explains the drop in 30-day rolling median values. Based on the public information (“Blender now has a much larger team of people working on core development”), we speculate that a sudden increase in the number of engineers may have caused a temporary increase in code velocity [Roosendaal, 2019]. The trend may have normalized in 2020 because of “unprecedented number of 108 new contributors” [Siddi, 2021].
Various data points and events can influence a project’s code velocity. Only a few of these potential variables are formally tracked. Decisions about feature development, project management, organizational challenges, or changing business priorities are not always documented and available to the public. We have only limited insight into all the confounding variables. To understand the potential
impact of metrics associated with project development, we mine the ones we can access. Our findings are displayed in Table 2.5.

Table 2.5: Median annual increase or decrease percentage in various code churn metrics per project. The scope is a code review period covered in Figure 2.1. We separate the roles of an author and a committer because they can differ. © 2023 IEEE.

<table>
<thead>
<tr>
<th>Project</th>
<th>Period</th>
<th>SLOC</th>
<th>Commit count</th>
<th>Distinct author</th>
<th>Distinct committer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blender</td>
<td>2014–2022</td>
<td>6.2%</td>
<td>−2.7%</td>
<td>13.5%</td>
<td>2.3%</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>2014–2022</td>
<td>3.5%</td>
<td>−1.2%</td>
<td>0.5%</td>
<td>−4.3%</td>
</tr>
<tr>
<td>LLVM</td>
<td>2014–2022</td>
<td>17.4%</td>
<td>6.8%</td>
<td>18.5%</td>
<td>15.6%</td>
</tr>
<tr>
<td>Mozilla</td>
<td>2017–2022</td>
<td>11.1%</td>
<td>−4.8%</td>
<td>−6.9%</td>
<td>−13.8%</td>
</tr>
</tbody>
</table>

The only common indicator between the projects is the continuous increase in the source lines of code. We focus on objective characteristics that we can mine from the source control management system and defect database. We look at the annual change in total source lines of code (calculated using scc [Boyter, 2022]), number of commits, number of distinct authors, and number of distinct committers. We use each project’s default branch from the original Git repository or an available GitHub read-only mirror.

We investigate the contents of defect-tracking databases associated with the projects we analyze. We find the metrics related to defects to be an unreliable indicator of the project’s workload. We find no formal rules related to defect counts and their implication for the development process. In addition, we do not observe that developers use the defects assigned to them as a primary list of work items.

2.3.5 Discussion

Our main finding is that there is no significant change in the trend for various code review periods. The lack of trend applies to all the projects regardless of the increase or decrease in commit count, number of distinct authors, and number of distinct committers. The median annual change in code base size for all the projects is an increase of 3–17%. At the same time, the 30-day median and all-time median for different code review periods change only in a second or third
decimal place (see Table 2.4). If anything, there is a minor decrease in the duration for time-to-first-response, time-to-accept, and time-to-merge.

The main finding is surprising. The conventional wisdom in software engineering is that communication and processes slow down with an increase in the project’s size and age. There could be multiple explanations for the results that we see. One possibility is that while the code complexity and team size continue to increase, engineers get more familiar and efficient while working in the project’s code base. With time the developers also build better interpersonal relationships that improve communication efficiency. It is reasonable to assume that development infrastructure also improves. Those factors can counteract the time spent on tasks such as complex debugging issues or the comprehension speed for the code sent for review.

Another possibility is that regardless of the project size, the code velocity stays in a specific range due to factors such as human nature. Engineers will get to reviewing the code when they get to it. Developers will spend a fixed amount of time on code reviews regardless of their familiarity with the code or other parallel priorities. Lehman’s fourth law (“Conservation of Organisational Stability”) means that “the work output of a project is independent of the amount of resources employed”) [M. Lehman, 1980]. The original statement relates to the rate of development productivity. It is reasonable to speculate that the exact underlying root cause is responsible for the lack of trend in code velocity.

The “flatness” or lack of a trend is visible in Figure 2.1 for FreeBSD, LLVM, and Mozilla. The Blender project has more variability in time-to-merge. The variability can be explained by Blender being the smallest of all the projects we investigate. For example, the median number of commits and distinct authors across the years in Blender are 8,589 and 89, respectively, while in LLVM, it is 29,555 and 592.

One more explanation is related to organizational self-correcting behavior. Each product we study has an informal or formal core team. The core team contains the most active or senior project members. Based on our experience with open-source software, the code velocity falling into a specific range can also result from core team members ensuring that code reviews get a timely response and the number of pending issues decreases.

2.3.6 Threats to validity

Our study is subjected to a specific category of threats. One threat relates to application of our findings in other contexts or external validity [Shull et al., 2008].
The projects that we investigate are all open-source software. The incentive structure in open-source software development is different from industrial projects. In addition, because of the code review granularity that we target, the project selection is limited to the ones that use Phabricator for code collaboration. We do not recommend generalizing these results without further replication in the target environment.

Another threat relates to internal validity. This threat type relates to the interpretation of the results and if correct conclusions are drawn from the data. We use the standard recommended nonparametric statistical apparatus to draw our conclusions. The metrics such as rolling 30-day means and median are widely used in quantitative finance as trend indicators [Hyndman, 2011]. We corroborate our findings by calculating 90-day and all-time medians that indicate a similar trend.

### 2.3.7 Data Availability

The Phabricator data, various R scripts that are used to perform statistical analysis, and relevant SQL queries are available on Figshare.\(^5\)

---

\(^5\)https://figshare.com/s/4558d92adc8d5d262bd6