# Revisiting the Applicability of the Pareto Principle to Core Development Teams in Open Source Software Projects

Kazuhiro Yamashita
Kyushu University, Japan
yamashita@posl.ait.kyushu-u.ac.jp

Shane McIntosh
McGill University, Canada
shanemcintosh@acm.org

Yasutaka Kamei
Kyushu University, Japan
kamei@ait.kyushu-u.ac.jp

Ahmed E. Hassan
Queen's University, Canada
ahmed@cs.queensu.ca

Naoyasu Ubayashi
Kyushu University, Japan
kamei@ait.kyushu-u.ac.jp

## ABSTRACT

It is often observed that the majority of the development work of an Open Source Software (OSS) project is contributed by a core team, i.e., a small subset of the pool of active developers. In fact, recent work has found that core development teams follow the Pareto principle — roughly 80% of the code contributions are produced by 20% of the active developers. However, those findings are based on samples of between one and nine studied systems. In this paper, we revisit prior studies about core developers using 2,496 projects hosted on GitHub. We find that even when we vary the heuristic for detecting core developers, and when we control for system size, team size, and project age: (1) the Pareto principle does not seem to apply for 40%-87% of GitHub projects; and (2) more than 88% of GitHub projects have fewer than 16 core developers. Moreover, we find that when we control for the quantity of contributions, bug fixing accounts for a similar proportion of the contributions of both core (18%-20%) and non-core developers (21%-22%). Our findings suggest that the Pareto principle is not compatible with the core teams of many GitHub projects. In fact, several of the studied GitHub projects are susceptible to the "bus factor," where the impact of a core developer leaving would be quite harmful.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Team size*

## General Terms

Measurement

## Keywords

Core development team, Pareto principle, Open source



*IWPSE'15*, August 30, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804366

## 1. INTRODUCTION

Understanding open source software (OSS) communities, i.e., the groups that are responsible for developing and maintaining an OSS system, is as important as understanding OSS systems themselves. By studying OSS communities, we accumulate knowledge about how these communities manage highly distributed development teams [18, 21]. Such knowledge enables the OSS development model to augment or replace development models in proprietary settings.

At the heart of OSS communities are *core developers*, i.e., the developers who take a leading role in the development and maintenance of a software project. For instance, Nakakoji *et al.* [19] state that core developers are *responsible for guiding and coordinating the development of an OSS project. Core Members are those people who have been involved with the project for a relative long time and have made significant contributions to the development and evolution of the system.* Mockus *et al.* [18] define core developers as the most productive developers who have made roughly 80% of the total contributions. Although these heuristics slightly differ, researchers agree that the impact that core developers have on a project is large.

Recent studies have shown that a small number of developers make a large proportion of the code contributions [5, 6, 18]. Moreover, it has been shown that the number of core developers follows the *Pareto principle* (a.k.a., the 80-20 rule), i.e., 80% of the contributions are produced by roughly 20% of the contributors [8, 17, 24].

Although the prior work makes important strides towards understanding core teams in OSS, the conclusions are drawn based on a small sample size (i.e., 1-9 studied systems). Therefore, in this paper, we set out to revisit how the Pareto principle applies to core teams in a large sample of 2,496 GitHub projects. We study GitHub projects because GitHub is one of the most popular social coding platforms, and many successful OSS systems are developed on GitHub (e.g., `Rails`[1]). Through analysis of the 2,496 GitHub projects, we address the following two research questions:

(RQ1) *Does the proportion of core developers of GitHub projects follow the Pareto principle?*
While the actual proportion of core developers varies depending on the heuristic of core developers that we

[1] `https://github.com/rails/rails`

**Table 1: An overview of the results of prior work.**

| Paper | Dataset | Result |
|---|---|---|
| Mockus *et al.* [18] | Apache and Mozilla | 10 to 15 developers performed 80% of the contributions. |
| Dinh-Trong and Bieman [5] | FreeBSD | 28 to 42 developers performed 80% of the contributions. |
| Koch and Shneider [17] | GNOME | 52 developers (out of 301 developers) performed 80% of the contributions. |
| Goeminne and Mens [8] | Brasero, Evince and Wine | 20% of developers performed 85%, 80% and more than 90% of the contributions in each project. |
| Robles *et al.* [24] | GNOME | The core group has been identified as the 20% most contributing committers. |
| Geldenhuys [6] | 9 OSS projects | 3%-9% of developers performed 80% of the contributions. |

use, 26%-58% of projects have core teams that are too small ($\leq$ 10% of active contributors) or 5%-28% have core teams that are too large ($\geq$ 30% of active contributors) to be considered compliant with the Pareto principle.

(RQ2) *Is there any difference between the contributions of core and non-core developers?*
Surprisingly, we find that the proportions of core and non-core developer activity are very similar when we normalize them by their contribution rates. For example, bug fixing activity accounts for 18%-20% of core developer contributions and 21%-22% of non-core developer contributions.

The main contributions of this paper are:

- A large-scale analysis of the core teams of 2,496 GitHub projects.
- A comparative analysis of three heuristics for identifying core developers.

**Paper organization.** The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes our heuristics for identifying core developers in more detail. Section 4 provides an overview of the studied GitHub projects. Section 5 describes the design and results of our case study. Section 6 discusses findings from our study. Section 7 discloses the threats to the validity of our study. Finally, Section 8 draws conclusions.

# 2. RELATED WORK

## 2.1 Proportion of Core Developers

Prior work has also analyzed the proportion of core developers in OSS projects. Table 1 provides an overview of the results of the prior work and the datasets that were analyzed. Mockus *et al.* [18] hypothesize that the open source development model would rely on a team of core developers who control the code base and that these core developers would create 80% or more of the new functionality. Furthermore, Mockus *et al.* argue that the core team would be no large than 10 to 15 people based on analysis of the Apache and Mozilla projects. Crowston *et al.* [3] compared three approaches to identify the core developers within 116 SourceForge projects using bug fixing activity. Although the results differ among the three studied approaches, all of the approaches indicate that the core developers make up a small fraction of the total number of contributors. Goeminne and Mens [8] found evidence for the Pareto principle in three activities (development, email discussion and bug tracker activity) in three OSS projects. Robles *et al.* [24] arrived at similar conclusions — the core team makes up roughly 20% of the contributing committers.

On the other hand, other studies arrive at contradictory conclusions. For example, Dinh-Trong and Bieman [5] replicated Mockus *et al.*'s work using data from the FreeBSD project, finding that 28-42 out of 161-265 developers perform 80% of the contributions. Koch and Shneider [17] find that 52 out of 301 developers make 80% of the contributions in the GNOME project. Through analysis of nine systems, Geldenhuys [6] find that the proportion of core developers does not comply with the Pareto principle.

Much of the prior work analyzes a small number of subject systems. Hence, we set out to analyze core teams in a large number of systems. More specifically, we formulate the following research question:

*(RQ1) Does the proportion of core developers of GitHub projects follow the Pareto principle?*

In addition to the size of core teams, Mockus *et al.* [18] hypothesized that a group, which is larger by an order of magnitude than the core team, will repair defects. From this hypothesis, we derive that non-core developers focus more on maintenance activity (e.g., bug fixing) than implementation activity. Goeminne and Bieman [8] showed that 2-6 out of the top 20 developers also contributed to plenty of the bug report and email discussions. However, to the best of our knowledge, the contribution activity of core and non-core developers have not been quantitatively compared. Hence, we formulate the following research question:

*(RQ2) Is there any difference between the contribution activity of core and non-core developers?*

## 2.2 Studies on GitHub

In recent years, GitHub has become a popular source of data for SE researches. Gousios *et al.* [10, 11] focus on the pull-based development process. They first answer basic questions about what the life cycle of a pull request is, and how prevalent the pull-based development process is [10]. In more recent work, Gousios *et al.* also study the impact that the pull-based development process has on integrators, who manage code contributions [11].

Dabbish *et al.* [4] conducted an interview with GitHub users to find out what inferences people make from GitHub transparency, and what the value of transparency for soft-

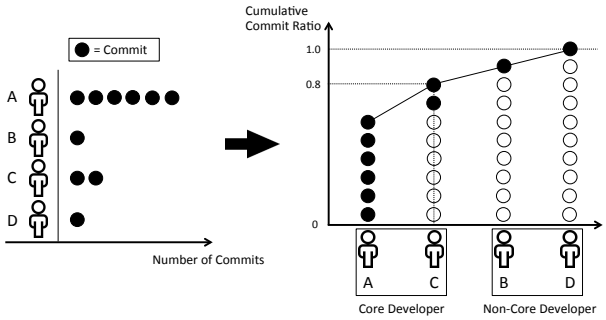



**Figure 1: Identifying core developers using an example project.**

ware development is. Their work reveals that four types of transparency-related and social inferences feed into three types of collaborative activities, such as project management. In the project management activities, the visibility of GitHub public repositories shifts development processes from being repository-focused to being owner-focused and contributor-focused.

Previous studies focused on the GitHub features. On the other hand, we focus on the proportion of core developers in GitHub projects.

# 3. HEURISTICS TO IDENTIFY CORE DEVELOPERS

In order to perform our study, we need to define heuristics to identify core developers. Inspired by previous studies, this paper explores three heuristics to identify core developer from the perspective of contributions as described below.

## 3.1 Commit-Based Heuristic

Several previous papers that have studied core developers [8, 18, 24] use the heuristic that defines core developers as those who produce roughly 80% of the total contributions. In this paper, we adopt this heuristic — after sorting the developers by their number of contributions in descending order, the core developers are those who have produced 80% of the project contributions, cumulatively.

For instance, Figure 1 shows an example project with four developers: A, B, C, and D. In order to determine core developers, we first sort the developers by the number of commits in descending order (A: 6, C: 2, B: 1 and D: 1). Next, we calculate the percentage of total commits that each developer has produced (A: 60%, C: 20%, B: 10%, and D: 10%). Then, we calculate the cumulative percentage (A: 60%, C: 80%, B: 90%, and D: 100%). Finally, we select core developers, one at a time, moving left to right, until we reach a cumulative percentage of 80%. In this example, A and C are identified as core developers.

In our algorithm, we do not handle the special case where there are some developers who have same number of commits on the border of core and non-core developers. We do not suspect that who we select to be a member of the core team should have a significant impact on the results, since: (a) these developers have produced the same number of contributions and (b) they are at the tail end of the core team contributions.

**Table 2: Finding self-identified mirror projects.**

| Category | Used regular expression [16] | #Projects |
|---|---|---|
| Mirror Of | $mirror\ of\ .*repo\|git\ repo\ of$ | 10 |
| Sourceforge | $sourceforge\|sf\backslash.net$ | 6 |
| Bitbucket | $bitbucket$ | 2 |
| Subversion | $\backslash W(svn\|subversion)\backslash W$ | 4 |
| Mercurial | $\backslash W(mercurial\|hg)\backslash W$ | 0 |
| CVS | $\backslash W cvs\backslash W$ | 0 |
| Total | - | 23 |

## 3.2 LOC-Based Heuristic

Similar to the commit-based heuristic, the LOC-based heuristic defines core developers according to the size of the contributions that they make [17, 18]. While we conduct our experiments using three size metrics, i.e., *the number of added lines*, *the number of deleted lines* and *the churn* (the sum of the number of added and deleted lines), we find that the results are similar across the three metrics. Therefore, to conserve space, we show only the results for churn in the remainder of the paper. Similar to commit-based heuristic, we identify core developers as those who cumulatively contribute 80% of the churn.

## 3.3 Access-Based Heuristic

Core developers can also be defined as those who have be given direct write access to the main VCS repository. For example, in projects like PostgreSQL, only core members can record changes directly in the main VCS repository — other contributors must convince core developers to record their changes on their behalf [7]. Hence, we can also identify core developers from a VCS access perspective.

In GitHub, project owners can grant write access to the project's main repository to other contributors. GHTorrent collects this information using the *collaborators* API and stores it in the *project_members* column [9]. According to the description of the collaborators API, the list includes all organization owners and users with access rights.[2] Since this list may include users who are members of an organization, but who did not contribute to a project, we define the access-based core developers as those who appear in the access list and have also made at least one commit.

Unfortunately, we find that roughly half of the studied projects do not use the access-based feature of GitHub. These projects are filtered out of our analysis when we use the access-based heuristic.

# 4. DATASET

In this section, we describe how we prepare the dataset of GitHub projects for our study. Figure 2 provides an overview of our dataset preparation steps.

We begin our study with the collection of GitHub project data that is available via GHTorrent [9]. However, GitHub hosts a large number of repositories, many of which are not software projects. Hence, we filter the GHTorrent data according to the suggestions of Kalliamvakou *et al.* [16]. We take three steps to create our dataset from the available GitHub projects. Initially, GHTorrent includes 8,510,504 repositories.

[2] `https://developer.github.com/v3/repos/collaborators/`

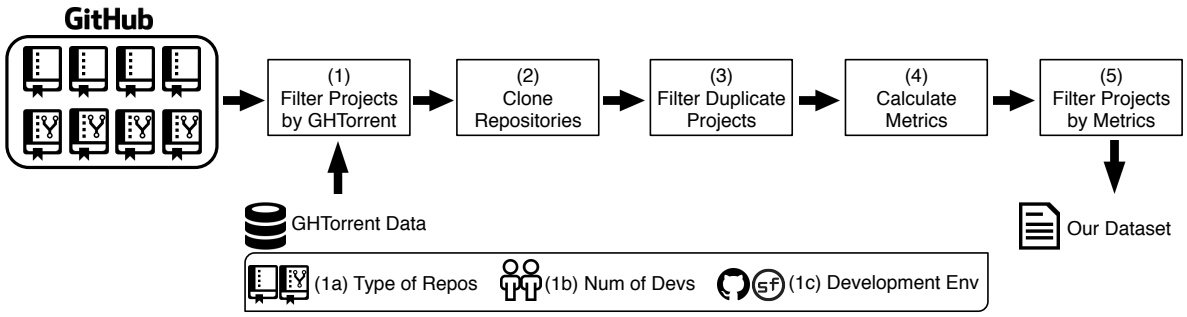



**Figure 2: An overview of our data extraction approach.**

## (1) Filter Projects by GHTorrent Data

**(1a) Type of Repository.** In GitHub, there are two types of repositories: *main repositories* and *fork repositories*. A fork is a working copy of a main repository. Forking a repository allows developers to freely experiment with changes without interfering with the ongoing development of the original project.[3] In GitHub, fork repositories can contribute changes back upstream to the main repositories that they are forked from by issuing pull requests. If the maintainers of the upstream repository agree with the changes that are proposed by a pull request, the request is accepted, and the changes are integrated into the main repository. As all accepted pull requests are stored in main repository, we only extract commits from the main repository, ignoring commits that only appear in forks.

**(1b) Number of Developers.** Two types of authorship data are recorded in Git repositories. The committer is the team member who recorded the changes in the repository using the `git commit` command. The author is the team member who produced the code change itself. In this study, we focus on the authors of the changes, ignoring the committer data, since the author is the team member who actually produced the changes, while the committer is the team member responsible for the integration work.

Furthermore, since projects with a small number of developers can easily achieve extreme core team proportions, we filter away projects that have too few developers (number of developers $< 10$).

**(1c) Development Environments.** In this study, we would like to investigate core developers especially in projects that are developed on GitHub. Kalliamvakou *et al.* [16] find that GitHub is not only a popular social coding platform, it also serves as a popular host for mirrored repositories.[4] Since such mirrored projects may not be developed in the same manner as projects on GitHub, we need to filter them out of our dataset. To do so, we heed the advice of Kalliamvakou *et al.* [16]:

1. Avoid projects that have a large number of committers who are not registered GitHub users.
2. Avoid projects that explicitly state that they are mirrors in their description.

To address item 1), we filter away projects where less than 90% of the committers are registered GitHub users. To address item 2), we filter away projects with descriptions that match the regular expressions listed in Table 2, as proposed by the prior work [16].

After applying the filters of steps (1a)-(1c), 4,618 projects remain in our dataset.

## (2) Clone Projects

Now that the number of projects has become manageable, we clone the selected repositories into our local environment to calculate the metrics that we use for our case study. Unfortunately, some of the projects that we select from the GHTorrent dataset are no longer available to be cloned (e.g., deleted repositories). Thus, we cannot include such projects in our dataset. Nonetheless, we could clone 4,154 projects.

## (3) Filter Duplicated Projects

Even after handling explicitly forked repositories, there are still some duplicate repositories hosted on GitHub (i.e., cloned and registered repositories that were not created using the GitHub fork feature). Such projects do not count as fork projects, but those projects have largely the same history as their originals. Since such projects will introduce noise in our dataset, we first detect them using the steps below, and then filter them out of our dataset.

We use the hashes of commits (SHAs) recorded in the Git repositories to identify duplicated projects. We consider any repositories that shares more than 70% of the same commit SHAs as a copied repository. We remove both repositories from our dataset because it is often difficult to determine which repository is the original one and which one is the copy.

After removing these repositories, 3,533 projects remain in our dataset.

## (4) Calculate Metrics from Repositories

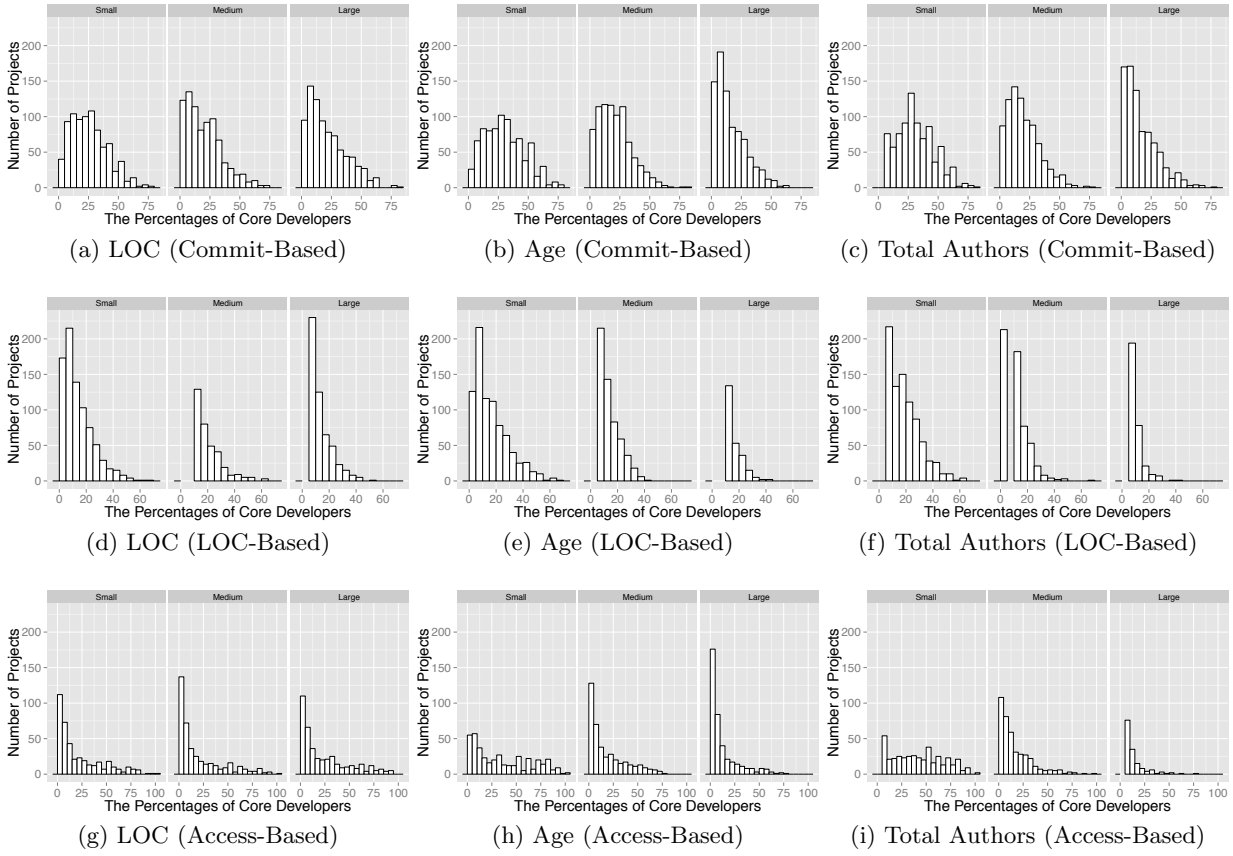
For the remaining projects, we calculate the metrics that are listed below in order to perform our case study.

**LOC.** We use *cloc*[5] to calculate LOC. Our LOC count does not include code comments or blank lines.

**Total Commits.** We count the number of commits by using the `git log` command with the `--no-merges` option.

**Total Authors.** We identify the unique authors by author name and email address, which we are able to extract from the commit logs. We use a tool[6] to disambiguate author

[3] `https://help.github.com/articles/fork-a-repo/`
[4] e.g., `https://github.com/apache`
[5] `http://cloc.sourceforge.net/`
[6] `https://github.com/bvasiles/ght_unmasking_aliases`




(a) LOC (Commit-Based) (b) Age (Commit-Based) (c) Total Authors (Commit-Based)

(d) LOC (LOC-Based) (e) Age (LOC-Based) (f) Total Authors (LOC-Based)

(g) LOC (Access-Based) (h) Age (Access-Based) (i) Total Authors (Access-Based)

**Figure 3: Distribution of Projects of Each Size Categories**

names and email addresses. We disambiguate names and email addresses of authors because some developers appear with slightly different forms [6].

**Age.** We calculate the age of a project (in days) by subtracting the time of the latest commit from the time of the initial commit.

## (5) Filter Projects by Metrics

We not only filter projects that have fewer than 10 developers, but similar to Bissyande *et al.* [1], we also filter projects that have fewer than 1,000 LOC.

Finally, we obtain a dataset that includes 2,496 GitHub projects for the commit-based and LOC-based core developer heuristic. Since 1,284 of these projects do not have the information that is needed to detect contributors with write access (*cf.* Section 3), only the remaining 1,212 projects are studied using the access-based heuristic.

# 5. STUDY RESULTS

In this section, we present the results of our study with respect to our two research questions. For each research question, we present our approach and our results.

## (RQ1) Does the proportion of core developers of GitHub projects follow the Pareto principle?

We begin our study by measuring the proportion of developers who are active enough to be considered core developers.

**Approach.** To address our first research question, we calculate the proportion of the development team that is considered to be part of the core team (*cf.* Section 3) of each studied project. Then, we use histograms to study the distribution of core team sizes in the studied projects.

The Pareto principle or the so-called "80-20 rule" states that 80% of the contributions are performed by roughly 20% of the contributors. In this study, similar to prior work [18, 23], we add a window of flexibility, considering projects where the core team proportion is $20\% \pm 10\%$ as being compliant with the Pareto principle. Indeed, Mockus *et al.* [18] showed that the core team proportions of modules in the Mozilla project are roughly 19%-25%. Moreover, Robles *et al.* showed that 10%-20% of developers produced more than 50% activities (in many cases as much as 90% or 95%).

We address RQ1 using two analyses. First, we analyze the distributions of proportions of core developers. Then, we split up the projects according to three confounding factors. Since the core team characteristics of smaller projects likely differs from those of larger projects, we divide the dataset into three strata (small, medium, and large) along three confounding factors (system size, team size and project age). We evenly divide the dataset accordingly, i.e., each stratum includes 832 projects for the commit-based and LOC-based heuristics, and each stratum includes 404 projects in the access-based heuristic. We then plot histograms of the core team proportions of projects in each of these nine strata. In this paper, we do not show the plots of overall distribution

**Table 3: The spread of projects among strata of project size and age.**

| Heuristic | Size Metrics | Stratum | Proportion of Core Developers | | |
|---|---|---|---|---|---|
| | | | 0%-10% | 10%-30% | 30%-100% |
| Commit-Based | LOC | Small | 143 (17%) | 411 (49%) | 278 (33%) |
| | | Medium | 264 (32%) | 389 (47%) | 179 (22%) |
| | | Large | 242 (29%) | 368 (44%) | 222 (27%) |
| | Age | Small | 94 (11%) | 359 (43%) | 379 (46%) |
| | | Medium | 203 (24%) | 447 (54%) | 182 (22%) |
| | | Large | 352 (42%) | 362 (44%) | 118 (14%) |
| | Total Authors | Small | 80 (10%) | 365 (44%) | 387 (47%) |
| | | Medium | 224 (27%) | 449 (54%) | 159 (19%) |
| | | Large | 345 (41%) | 354 (43%) | 133 (16%) |
| | General | | 649 (26%) | 1,168 (47%) | 679 (27%) |
| LOC-Based | LOC | Small | 403 (48%) | 367 (44%) | 62 (7%) |
| | | Medium | 487 (59%) | 304 (37%) | 41 (5%) |
| | | Large | 557 (67%) | 253 (30%) | 22 (3%) |
| | Age | Small | 354 (42%) | 376 (45%) | 102 (12%) |
| | | Medium | 501 (60%) | 316 (38%) | 15 (2%) |
| | | Large | 592 (71%) | 232 (28%) | 8 (1%) |
| | Total Authors | Small | 227 (27%) | 497 (60%) | 108 (13%) |
| | | Medium | 502 (60%) | 315 (38%) | 15 (2%) |
| | | Large | 718 (86%) | 112 (13%) | 2 (0.2%) |
| | General | | 1,447 (58%) | 924 (37%) | 125 (5%) |
| Access-Based | LOC | Small | 192 (48%) | 100 (25%) | 112 (28%) |
| | | Medium | 211 (52%) | 92 (23%) | 101 (25%) |
| | | Large | 177 (44%) | 98 (24%) | 129 (32%) |
| | Age | Small | 115 (28%) | 94 (23%) | 195 (48%) |
| | | Medium | 202 (50%) | 107 (26%) | 95 (24%) |
| | | Large | 263 (65%) | 89 (22%) | 52 (13%) |
| | Total Authors | Small | 60 (15%) | 87 (22%) | 257 (64%) |
| | | Medium | 191 (47%) | 143 (35%) | 70 (17%) |
| | | Large | 329 (81%) | 60 (15%) | 15 (4%) |
| | General | | 580 (48%) | 290 (24%) | 342 (28%) |

**Table 4: Distributions of projects according to the number of core developers.**

| Number of Core Developers | 1-9 | 10-15 | 16-20 | 21-50 | 51-100 | 101- |
|---|---|---|---|---|---|---|
| Commit-Based | 1,924 (77%) | 273 (11%) | 98 (4%) | 137 (5%) | 17 (0.7%) | 47 (2%) |
| LOC-Based | 2,397 (96%) | 57 (2%) | 15 (0.6%) | 13 (0.5%) | 4 (0.1%) | 10 (0.5%) |
| Access-Based | 1,036 (85%) | 128 (11%) | 24 (2%) | 24 (2%) | 0 (0%) | 0 (0%) |

to conserve space because we find that the distributions of the medium strata follow the same trends as the overall distributions.

**Results.** Figure 3 shows the core team distributions of the studied projects. Table 3 shows the exact numbers of projects of each category and percentile. In Table 3, the gray colored columns show the Pareto-compliant range.

**Contrary to prior results, we find that the core team size of projects distributes broadly.** Figure 3 and Table 3 show that the distributions are different according to the heuristic. Indeed, unlike prior work [8, 17, 24], we find that there are many projects that fall outside of our range of Pareto compliance (10%-30%).

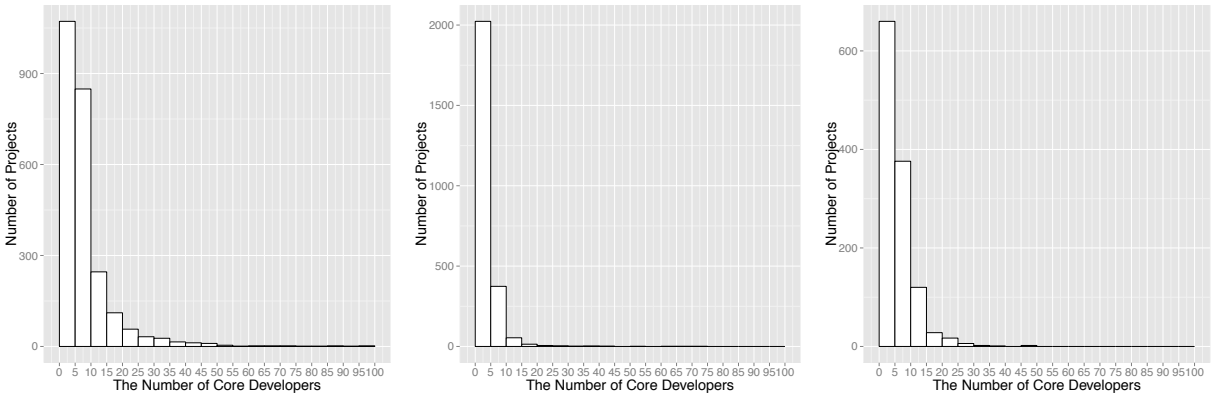
When we focus on each heuristic and confounding factor, we observe the following trends.

Commit-Based: Table 3 shows that, irrespective of the stratum, 43%-54% of the studied projects are Pareto compliant. When controlling for project age and team size, we find that the number of projects with the smallest core team size (i.e., 0%-10%) increases as we shift from the smallest to largest strata. On the other hand, this trend is not as extreme in the system size strata. Therefore, we conclude that project age and team size have a larger impact on the core team proportion than system size does.

LOC-Based: The LOC-based heuristic is more right skewed than the commit-based heuristic. Similar to the commit-based heuristic, Table 3 shows that the right skew increases as the system size increases. Moreover, the total number of authors seems to impact to the core team proportion because the difference between small and large stratum is the largest among the three studied metrics.

Access-Based: Figure 3 shows that the distributions of the access-based heuristic are similar to those of the LOC-based heuristic. However, there are more projects that fall in the 30%-100% range for the access-based heuristic than the LOC-based heuristic. Similar to the commit-based heuristic (Table 3), age and team size also appear to have an impact on the core team proportion of the access-based heuristic.

Figure 4 shows the number of core developers. In Figure 4, the x-axis shows the number of core developers and the y-axis




(a) Commit-Based (b) LOC-Based (c) Access-Based

**Figure 4: The distribution of projects according to the number of core developers.**

shows the number of projects. Table 4 shows the breakdown of projects stratified by the number of core developers.

From the perspective of core team size, we find support for the findings of prior studies [5, 17, 18]. Mockus *et al.* argue that if the core team uses only an informal means of coordinating, the group will be no larger than 10-15 people [18]. Conversely, Dinh-Trong and Bieman [5] find that 28-42 developers provide 80% of the contributions in the FreeBSD project. Koch and Schneider [17] find that 52 developers provide 80% of the contributions in GNOME project.

**88%-98% of projects have fewer than 16 core developers.** Unlike the proportion of core developers, the distributions of the number of core developers are similar across the studied heuristics. Indeed, Table 4 shows that 88%-98% of the studied GitHub projects have fewer than 16 core developers.

We further analyze the 2%-12% of projects that have more than 15 core developers to find out what kind of projects have larger core teams. When using the commit-based heuristic, 275 out of the 299 projects that have more than 15 core developers are categorized in large stratum of total authors and the remaining 24 projects are in medium stratum. When using the LOC-based heuristic, 41 of the 42 projects are in large stratum of total authors and the remaining one project is in medium stratum. When using the access-based heuristic, 27 of the 48 projects are in large stratum of total authors, and 20 of the remaining 21 projects are in medium stratum. These observations indicate that most of the projects that have many core developers also tend to a larger pool of contributors than the other projects.

*Contrary to prior work, we find that there are several projects that have larger or smaller core team proportion than we consider to be compliant with the Pareto principle. Moreover, we find that most projects have 15 or fewer members of the core team.*

## (RQ2) Is there any difference between the contribution activity of core and non-core developers?

To address this RQ, we compare the types of contributions that are performed by core and non-core developers.

**Table 5: Keywords used to classify commits [12].**

| **Development** | |
|---|---|
| Activity Type | Keywords |
| *Forward Engineering* | implement, add, request, new, test,start, includ, initial, introduc, creat, increas |
| **Maintenance** | |
| Activity Type | Keywords |
| *Reengineering* | optimiz, adjust, update, delet, remov, chang, refactor, replac, modif, (is, are) now, enhance, improv, design change, renam, eliminat, duplicat, restructur, simplif, obsolete, rearrang, miss, enhanc |
| *Corrective Engineering* | bug, fix, issue, error, correct, proper, deprecat, broke |
| *Management* | clean, license, merge, release, structure, integrat, copyright, documentation, manual, javadoc, comment, migrat, repository, code review, polish, upgrade, style, formatting, organiz, TODO |

**Approach.** Previous studies have explored the purposes of changes [12, 14, 20]. In this study, we adopt Hattori and Lanza's approach to identify the purpose of changes. Hattori and Lanza [12] proposed a lightweight approach to classify each commit into development or maintenance activities based on the accompanying commit messages. They defined four main activities: *forward engineering* as development activity; and *reengineering*, *corrective engineering* and *management* as maintenance activity. They also provide keywords that are indicative of the type of activity (Table 5). *Forward engineering* activities implement new requests and add new features. *Reengineering* activities are related to refactoring, redesign and other actions to enhance the quality of the code. *Corrective engineering* activities fix defects. *Management* activities are other general maintenance activities that are not related to system functionality, such as code reformatting and documentation.

To ensure that the classification provided by Hattori and

**Table 6: Developer Activity**

| Type of Activity | Commit-Based | | LOC-Based | | Access-Based | |
|---|---|---|---|---|---|---|
| | Core | Non-Core | Core | Non-Core | Core | Non-Core |
| Forward Engineering | 15% | 18% | 16% | 18% | 17% | 16% |
| Reengineering | 29% | 30% | 29% | 30% | 24% | 30% |
| Corrective Engineering | 20% | 21% | 20% | 22% | 18% | 21% |
| Management | 14% | 13% | 14% | 13% | 12% | 15% |
| Empty | 0.1% | 0.1% | 0.1% | 0.1% | 0.3% | 0.1% |
| Unknown | 22% | 17% | 22% | 17% | 30% | 19% |
| Total #of Commits | 4,692,063 | 1,054,460 | 4,739,121 | 1,007,402 | 931,265 | 1,934,196 |

Lanza is sufficient for our dataset, we manually analyze a randomly selected sample of 384 commit comments. The sample is selected such that it provides a confidence level of 95% with a confidence interval of ±5%. The manual analysis reveals that some commits have an empty commit comment. We classify such commits as *empty*.

Hattori and Lanza's approach searches for keywords in commit messages in the following order: empty comments, management, reengineering, corrective engineering and forward engineering. The commit comments of so-called tangled changes [13] can match multiple purpose keyword types. For example, a developer can clean up code and fix a bug within one commit. For these commits, the approach classifies the commit according to the keyword that is found first (e.g., the commit described above is classified into reengineering activity). The commits that could not be classified into any of the classes are marked as *unknown*.

Using Hattori and Lanza's approach, we classify and compare the distributions of activities of core and non-core developers. In total, our commit-based and LOC-based heuristic datasets includes 5,746,523 commits, and our access-based one contains 2,865,461 commits.

**Results.** Table 6 shows the distribution of activities of core and non-core team members. Interestingly, the total number of commits that are contributed by core and non-core team members are very similar when we use either commit-based and LOC-based heuristics. On the other hand, the access-based heuristic shows that the number of commits of core developers is less than that of non-core developers. In this study, we only consider the authors of commits. Hence, this discrepancy between core and non-core contributions might show that many of the access-based core developers focus on integration work rather than writing code.

**The proportions of contribution activity of core and non-core developers are similar.** Irrespective of the core team heuristic, we find that the distributions of activities are very similar. Reengineering accounts for the largest proportion of activity for both core and non-core developers, with proportions ranging between 24%-30%. In the other type of activities, the difference between the proportion of activity of core and non-core developers is at most 6 percentage points. Therefore, we conclude that the difference in activity proportions between core and non-core is negligible.

*The proportions of contribution activity of core and non-core developers are similar.*

# 6. DISCUSSION

## 6.1 The Bus Factor

We find that more than half of the studied projects have a core team comprised of (at most) 20% of the pool of active developers and more than 88% of the studied projects have a core team of (at most) 15 developers. These results indicate that many projects have a low *bus factor* [2, 22, 25], i.e., face the risk of key personnel leaving the project. Ye and Kishida [26] find that development of GIMP was once halted because a key core developer left the project. To avoid such cases, projects must share knowledge among developers.

On the other hand, similar to the work of Dinh-Trong and Bieman [5], we find that there are projects that have large core teams. In this study, we just show the distribution and do not investigate each of the projects deeply. In future work, we plan to conduct a deeper analysis of projects with large core teams. For example, investigating whether or not such projects have well-defined mechanisms for developer promotion rather than the informal arrangements that Mockus *et al.* [18] hypothesized could yield fruitful results.

## 6.2 Core and Non-core Developer Activity

Prior work [18] hypothesized that a group larger by an order of magnitude than the core team will repair defects. If the hypothesis is true, we assumed that the proportion of maintenance activity of non-core developers is large. However, our results show that both types of developers have similar proportions of development activities. Furthermore, when we consider the number of corrective engineering commits, the number of the commits by core developers is much larger than that by non-core developers.

Our results may be a characteristic of the GitHub development environment. With the growth of social coding platforms (e.g., GitHub), the nature of core teams in modern OSS projects may have changed. For example, GitHub projects boast a higher rate of acceptance for contributions than the OSS projects of the past did. Indeed, while Jiang *et al.* [15] find that only 33% of contributions are eventually integrated into the Linux kernel (one of the largest OSS projects, which mainly developed by outside of GitHub), Gousios *et al.* [10] find that 84% of contributions are eventually integrated into GitHub projects.

## 6.3 The Impact of Thresholds

In this study, we filter projects to remove immature software projects by using some thresholds, i.e., the total authors and LOC (*cf.* Section 4). As such, our results may be sensitive to these thresholds. To check for threshold sensitivity, we re-apply our analysis using other threshold values (total

**Table 7: The proportion of projects that are Pareto compliant when we use other threshold values.**

| Metric | Threshold | #ofProjects | Proportion |
|---|---|---|---|
| Total Authors | 5 | 2,526 | 46% |
| | 20 | 1,664 | 49% |
| LOC | 500 | 2,685 | 46% |
| | 2,000 | 2,220 | 47% |

authors = 5, 20 and LOC = 500, 2,000) and discuss changes to our results below.

Table 7 shows the proportion of projects that are Pareto compliant when we vary the thresholds. Irrespective of the threshold, similar to our results in Section 5, we observe that more than half of projects are not Pareto compliant. These results suggest that while our results slightly vary when the thresholds change, the main conclusions are not heavily impacted.

# 7. THREATS TO VALIDITY

## 7.1 Construct Validity

In this paper, we adopt three heuristics to identify core developers. The commit-based and LOC-based heuristics are based on the amount of contribution to the product. Even though there are a lot of metrics that can capture contribution units, the amount of contribution is one of the most basic metrics that is used to identify core developers. Moreover, previous studies that focus on core contributors [5, 6, 8, 17, 18, 24] also conduct their analysis from the perspective of the amount of contribution. Therefore, we feel that these heuristics are appropriate for our context.

On the other hand, the access-based heuristic does not depend on the amount of contribution. However, the access-based definition is also one of the most basic indicators of core developers. Indeed, the developers who have write access to the main repository have enough knowledge about the product to manage other developers' contributions.

## 7.2 Internal Validity

Our results for RQ1 are dependent on our heuristics for identifying core developers. In this study, we used 80% of the total contributions as our threshold for identifying core developers, since this threshold was also used by previous studies [5, 6, 8, 17, 18, 24]. While we begin a threshold sensitivity analysis in Section 6, we plan to perform a carefully controlled sensitivity analysis in future work.

Furthermore, our analysis is time-agnostic. Since development teams are changing over time, the number of core developers may vary as well. We plan to conduct a temporal analysis of core teams in future work.

## 7.3 External Validity

In this study, we filter away projects that have less than 10 developers or less than 1,000 LOC to remove projects that are immature [1, 16]. Therefore, our results may not generalize to legitimate software projects with a small number of contributors.

# 8. CONCLUSION

Open Source Software (OSS) projects depend heavily on core developers, i.e., team members that produce 80% of the contributions to a project. Prior studies have found that core development teams tend to follow the Pareto principle (a.k.a., the 80-20 rule), i.e., 80% of the contributions are produced by roughly 20% of the contributors. However, these prior studies were performed on small samples of systems. With the recent growth in popularity of the social coding paradigm, a plethora of data is becoming available for researchers to explore core team dynamics within. Therefore, we revisit the analyses of previous work on a large sample of GitHub projects.

To that end, in this paper, we study core development teams on GitHub. Through a case study of 2,496 GitHub projects, we observe that:

- The core teams of many GitHub projects are not compliant with the Pareto principle.
- While some GitHub projects have core teams that are too large to be Pareto compliant, many more have very small core teams, consisting of fewer than 10% of the pool of contributors.
- Core and non-core developers participate in maintenance and future development activities in similar proportions.

# 9. ACKNOWLEDGEMENTS

This research was partially supported by JSPS KAKENHI Grant Numbers 15H05306, the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers.

# 10. REFERENCES

[1] T. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE)*, pages 188–197, Nov 2013.

[2] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. Assessing the bus factor of git repositories. In *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503, 2015.

[3] K. Crowston, K. Wei, Q. Li, and J. Howison. Core and periphery in free/libre and open source software team communications. In *Proc. Hawai'i Int'l Conf. on System Science (HICSS)*, 2006.

[4] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proc. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 1277–1286, 2012.

[5] T. Dinh-Trong and J. Bieman. The freebsd project: a replication case study of open source development. *IEEE Trans. on Software Engineering*, 31(6):481–494, June 2005.

[6] J. Geldenhuys. Finding the core developers. In *Proc. of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 447–450. IEEE Computer Society, Sept. 2010.

[7] D. M. German. A study of the contributors of postgresql. In *Proc. Int'l Workshop on Mining Software Repositories (MSR)*, pages 163–164, 2006.

[8] M. Goeminne and T. Mens. Evidence for the pareto principle in open source software activity. In *Joint Porc. the 1st Int'l Workshop on Model Driven Software Maintenance and 5th Int'l Workshop on Software Quality and Maintainability*, pages 74–82, 2011.

[9] G. Gousios. The ghtorrent dataset and tool suite. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 233–236, 2013.

[10] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 345–355, 2014.

[11] G. Gousios, A. Zaidman, M.-A. Storey, and A. v. Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, 2015. To appear.

[12] L. Hattori and M. Lanza. On the nature of commits. In *Proc. Int'l Conf. on Automated Software Engineering (ASE) - Workshops*, pages 63–71, Sept 2008.

[13] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[14] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 99–108, 2008.

[15] Y. Jiang, B. Adams, and D. German. Will my patch make it? and how fast? case study on the linux kernel. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 101–110, May 2013.

[16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 92–101, 2014.

[17] S. Koch and G. Schneider. Effort, cooperation and coordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27–42, 2002.

[18] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. on Software Engineering and Methodology*, 11(3):309–346, 2002.

[19] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 76–85, 2002.

[20] R. Purushothaman and D. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. on Software Engineering*, 31(6):511–526, 2005.

[21] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[22] F. Ricca, A. Marchetto, and M. Torchiano. On the difficulty of computing the truck factor. In *Product-Focused Software Process Improvement*, volume 6759 of *Lecture Notes in Computer Science*, pages 337–351. 2011.

[23] G. Robles, J. Gonzalez-Barahona, and I. Herraiz. Evolution of the core team of developers in libre software projects. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 167–170, May 2009.

[24] G. Robles, S. Koch, J. M. González-Barahona, and J. Carlos. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *Proc. the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, pages 51–55, 2004.

[25] M. Torchiano, F. Ricca, and A. Marchetto. Is my project's truck factor low?: Theoretical and empirical considerations about the truck factor threshold. In *Proc. Int'l Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 12–18, 2011.

[26] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 419–429, 2003.