An Empirical Study of Metric-based Comparisons of Software Libraries

Fernando López de la Mora
University of Alberta, Canada
lopezdel@ualberta.ca

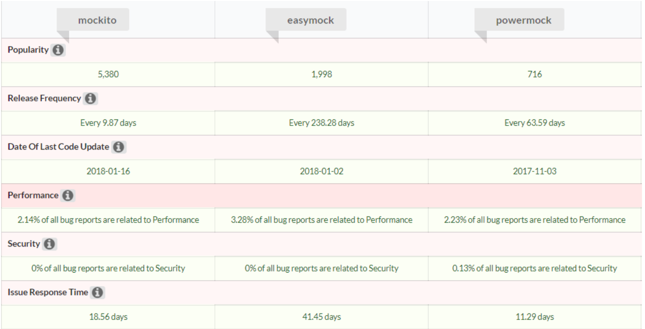
Sarah Nadi
University of Alberta, Canada
nadi@ualberta.ca

ABSTRACT

BACKGROUND: Software libraries provide a set of reusable functionality, which helps developers write code in a systematic and timely manner. However, selecting the appropriate library to use is often not a trivial task. AIMS: In this paper, we investigate the usefulness of software metrics in helping developers choose libraries. Different developers care about different aspects of a library and two developers looking for a library in a given domain may not necessarily choose the same library. Thus, instead of directly recommending a library to use, we provide developers with a metric-based comparison of libraries in the same domain to empower them with the information they need to make an informed decision. METHOD: We use software data analytics from several sources of information to create quantifiable metric-based comparisons of software libraries. For evaluation, we select 34 open-source Java libraries from 10 popular domains and extract nine metrics related to these libraries. We then conduct a survey of 61 developers to evaluate whether our proposed metric-based comparison is useful, and to understand which metrics developers care about. RESULTS: Our results show that developers find that the proposed technique provides useful information when selecting libraries. We observe that developers care the most about metrics related to the popularity, security, and performance of libraries. We also find that the usefulness of some metrics may vary according to the domain. CONCLUSIONS: Our survey results showed that our proposed technique is useful. We are currently building a public website for metric-based library comparisons, while incorporating the feedback we obtained from our survey participants.

1 INTRODUCTION

Software libraries provide ready-to-use functionality through their *Application Programming Interfaces* (APIs), which helps developers build reliable systems more efficiently. For a given domain such as databases or cryptography, it is common to find more than one available library that can perform the desired functionality. While it is always good for client developers (i.e., developers who want to use a library) to have several options to choose from, it is not always clear which of these libraries is best suited for a developer's needs. Making an inappropriate selection could have negative consequences. For example, developers may decide to migrate to a different library for reasons such as performance problems [14].



	mockito	easymock	powermock
Popularity ℹ			
	5,380	1,998	716
Release Frequency ℹ			
	Every 9.87 days	Every 238.28 days	Every 63.59 days
Date Of Last Code Update ℹ			
	2018-01-16	2018-01-02	2017-11-03
Performance ℹ			
	2.14% of all bug reports are related to Performance	3.28% of all bug reports are related to Performance	2.23% of all bug reports are related to Performance
Security ℹ			
	0% of all bug reports are related to Security	0% of all bug reports are related to Security	0.13% of all bug reports are related to Security
Issue Response Time ℹ			
	18.56 days	41.45 days	11.29 days

Figure 1: Excerpt of a metric-based comparison of libraries

Developers often use Q&A websites such as Stack Overflow to inquire about libraries that are appropriate for a given task, leading to discussions of how similar libraries compare to each other. Although these forums inform readers on some *aspects*, i.e., features or characteristics [33], of these libraries, other relevant aspects are usually not covered. Furthermore, users' opinions are typically based on their own experience rather than comparable data such as software *metrics*, a term which we use to refer to quantifiable data that describes or relates to an aspect of a library.

In a recent ICSE NIER paper, we proposed the idea of a metric-based comparison of software libraries [7]. Similar to the idea of online shopping where, for example, customers can compare different specifications of a computer monitor, we argued that libraries could also be compared based on quantifiable metrics, allowing developers to make their own informed decision about which library is best suited to their needs. Our long-term vision was to create a continuous surveying and crowd-sourced website that provides developers with metric information for library selection purposes. In that paper, we surveyed the literature to identify metrics that can be used to compare libraries, but we did not extract the data needed to compute all metrics, nor did we evaluate the idea. In this paper, we implement and assess our proposed idea. Our goal is to evaluate the usefulness of a metric-based comparison of software libraries and to obtain developers' feedback about which library metrics matter to them. To accomplish this objective, we implement an actual metric-based comparison and show it to participants as part of a survey. Specifically, our survey is designed to answer the following research questions:

- RQ1. Is a metric-based comparison of libraries useful for developers when selecting a software library to use?
- RQ2. Which metrics influence developers' library selections when comparing software libraries?

PROMISE'18, October 10, 2018, Oulu, Finland

ACM ISBN 978-1-4503-6593-2/18/10…$15.00
https://doi.org/10.1145/3273934.3273937

- RQ3. Does the library domain affect metric usefulness?

To answer our research questions, we first implemented a tool chain to compute our selected metrics. We calculate these metrics by mining a combination of sources, namely version control systems, issue tracking systems, and Q&A websites. As subject libraries for our metric-based comparisons, we collected 34 open-source Java libraries from 10 popular domains. We consolidate the metric information that we extract in a website where we present side-by-side comparisons of libraries of the same domain. Figure 1 shows an example of these comparisons for the *mocking* domain.

We use our metric-based comparison to survey 61 Java developers from diverse backgrounds, including professional developers, students, and researchers. Our results show that (i) a metric-based comparison of software libraries is useful to most developers when choosing libraries, (ii) *Performance*, *Popularity*, and *Security* are the most useful metrics for developers when selecting a library, and (iii) some metrics that developers care about are independent of the domain (e.g. Popularity), while the usefulness of some metrics is not relevant in certain domains (e.g. Release Frequency). We also present an extensive discussion of 147 free-text comments from participants, as they provide a valuable source of information that can be used to design better decision support for library selection.

To summarize, in comparison to our NIER paper, the new contributions of this paper are:

- A survey of 61 software developers to determine comparison metrics desired by developers, and the general usefulness of a metric-based comparison of software libraries for library selection purposes.
- A concrete open-source implementation [2] of a library metric-based comparison for 34 libraries from 10 domains that we use in the survey.
- A discussion of 147 free-text comments that demonstrate the factors and metrics that influence developers' library selections, and the implications of these comments. We plan to use the collected feedback from our survey to develop a public website for library comparisons.

2 RELATED WORK

We divide related work into two parts: (1) assistance for library/API selection and (2) extracting software metrics for different purposes.

Library Selection Assistance. Assisting developers in selecting libraries, library versions, and API elements based on collected data has been researched in previous work. Most recently, Uddin et al. [33] mined opinions from Stack Overflow posts to summarize APIs based on aspects such as documentation and performance. While some of our metrics are related to aspects they discuss, we are extracting quantifiable data from open-source repositories, issue-tracking systems, and Stack Overflow, instead of focusing on opinions from one source of information. Thung et al. [32] used data mining techniques to recommend libraries based on the currently used libraries of a client project. Rahman et al. [26] used questions and answers from Stack Overflow to automatically recommend APIs for a given task described in natural language. Teyton et al. [30] proposed a mining technique to identify appropriate alternatives to replace existing libraries in a project. Hora et al. [13] ranked API elements based on popularity and migration data mined from open-source repositories. Mileva et al. [20] mined popularity information from open-source repositories to provide recommendations of API elements. Mileva et al. [19] created a tool that assists developers in selecting the most stable version of a library based on usage trends of library versions, including the number of projects using a given library version. Unlike the work above, we combine several quantifiable metrics in our comparison of libraries. Furthermore, we do not provide a recommender system; instead, we empower developers with information about different aspects of a library, and leave them to make the choice based on their needs.

Software Metrics. Researchers have collected software metrics with purposes other than providing assistance for library selection. Linares-Vazquez et al. [17] computed the number of bug fixes and changes in the public elements of Android APIs and correlated these metrics with the success of client mobile applications. Teyton et al. [31] identified library migrations in over 8,000 Java open-source software projects by analyzing changes in library dependencies across different versions of a project. Kabinna et al. [14] analyzed bug reports related to library replacements and their respective version control history to study migrations of logging libraries. Past research has also applied sentiment analysis on information found in software repositories. Guzman et al. [12] applied sentiment analysis on commit comments from Github projects, and found negative and positive sentiments linked to different characteristics of a project. Borges et al. [4] investigated the factors that affect popular Github repositories and used the number of repository stars as proxy for popularity. We make additional references to software metrics in Section 3.

3 METRICS FOR LIBRARY COMPARISON

Uddin et al. [33] found that developers who read discussion forums prefer to see information about specific API aspects including performance, security, compatibility, community, and bugs. Accordingly, we design 9 metrics that are related to these aspects for our first implementation of a metric-based comparison of libraries. We selected metrics for which we can accurately extract and update their data via an automated methodology. These metrics are shown in a table format where rows refer to metrics, column headers contain libraries of a given domain, and single cells have the metric data for libraries, as shown in Figure 1. Users could then compare the metric data and decide which library is more convenient to them according to their needs. In the rest of this section, we provide a definition and intuition for each metric we use, as well as its respective extraction technique. We use the *mockito* library as a working example to explain the final metrics users see. All our code and extracted data can be found online [2].

3.1 Popularity

Definition and Intuition. Developers may simply want to use what the majority of developers are using. We define *Popularity* as the number of client projects using a given library. This approach has been frequently used as a proxy for popularity in the literature [13, 20]. Instead of using popularity of API elements such as classes of a library, we focus on the number of projects using any API of a given library.

Extraction Methodology. To obtain Popularity data, we use Boa [8], a large-scale mining infrastructure. To count the number of client projects of a given library, we wrote a script that searches the latest snapshot of Java files in a project and looks for Java *import* statements that include the general package of a library in these files. If at least 1 such statement is found, we assume the project uses the library. For example, mockito's popularity is 5,380, which represents its number of client projects.

3.2 Release Frequency

Definition and Intuition. Developers may be interested in knowing how often a library is updated, as new releases usually contain bug fixes and added functionality. We define library *Release Frequency* as the average time between two consecutive releases of a library. Software release cycles have been previously studied to understand their impact on quality assurance metrics [6, 15].

Extraction Methodology. We use the Github API to extract the *git* tag information found in the repository of the library, since each tag usually represents a release version of the software [1]. We did not use the Github release API as we found that it did not provide us with the full list of releases. For each tag, we extract the commit associated with the tag and use the commit date as the release date of the library version. To calculate Release Frequency, we find the average of the time difference between each two consecutive releases of the library. As an example, the Release Frequency of mockito is 9.87 days, which represents the average time between consecutive releases.

3.3 Issue Response Time and Issue Closing Time

Definition and Intuition. Potential clients may want to know if a given library's developers are helpful and if there is an active community around it. One quantifiable way to check this is to see how quickly reported issues are replied to and resolved. We refer to *Issue Response Time* as the average time that it takes to receive a comment once a bug report has been opened. Similarly, we define *Issue Closing Time* as the average time that it takes to close a bug report since it was originally opened. Bug-fix times have been researched in the literature. Examples include studying the use of bug report attributes such as severity, priority, and assignee to predict fixing times [9] and studying the relation of comment sentiments in JIRA bug reports with the fixing time [23].

Extraction Methodology for Issue Response Time. We use the Github API to obtain all issues found in a given library whose issue tracking system is hosted on Github. If a library uses JIRA to track issues, we export all issues as an XML file and parse the contents. For each issue, we extract its creation date, as well as the date of the first comment on the issue. The issue response time is then calculated as the difference between the creation date and the date of the first comment. We then calculate the issue response time of the library as the average of response times for all considered issues. Note that we discard issues that have no comments as we care about the average time the library community took to reply to issues. We use the first comment made by any user other than the original poster to calculate response time, as knowledgeable users who may not be major contributors to the project often provide useful suggestions or feedback on the issue. Our extracted Issue Response Time for mockito is 18.56 days.

Extraction Methodology for Issue Closing Time. We again use the Github API to extract all issues posted in the library's repository for Github-hosted tracking systems. Similarly, we export all issues as an XML file for libraries whose issue tracking systems are on JIRA. As this metric focuses on closing times, we discard issues which are not in a closed state. For each issue, we extract its creation and closing dates. For bug reports hosted in JIRA, we use the resolved date as the closing date for issues that had a closed status, as we found that JIRA does not provide an explicit closing date in its issue reports. The closing time of each issue is the difference between the creation and closing dates. Finally, to calculate the issue closing time metric for the library, we use the average of the closing time of all closed issues. We extract an Issue Closing Time of 70.84 days for mockito.

3.4 Backwards Compatibility

Definition and Intuition. A library is said to be *backwards compatible* if client projects can upgrade to a more recent version of the library without having to modify code that used the library's APIs. Backwards compatibility problems in libraries have been studied by Mostafa et al. [21] and by Xavier et al. [35]. Libraries that often break existing code may result in more maintenance work from client developers, so we believe that developers may want to see information about backwards compatibility when choosing a library.

Extraction Methodology. We use Xavier et al.[35]'s diff tool that analyzes two source code versions of a given library to detect changes to three types of API elements: type, field, and method. For types, breaking changes consist of type removal, visibility loss of the type (e.g. from public to private), and changes in its base type. For fields, field removals, modifications in the field's type, visibility loss, or different default values are considered breaking changes. Breaking changes for methods can be due to method removals, visibility loss, changes in its return type, parameter list changes, and exception list changes. We use this diff tool to count the number of breaking changes between two consecutive versions of a library for all releases R_1 to R_{n-1}, where n is the latest release of a library. Similar to our Release Frequency methodology, we obtain releases of a library by collecting git tags. Finally, we calculate the average of the resulting number of breaking changes across all analyzed releases. We obtain an average of 165.16 breaking changes per release for mockito.

3.5 Performance and Security

Definition and Intuition. Performance of a library refers to how efficient and optimized its code is. *Security* of a library shows its ability to handle sensitive information without compromising the data and its robustness against attacks. Developers may want to avoid libraries with many performance and/or security problems.

Extraction Methodology. To obtain quantifiable data about non-functional attributes of a library such as performance or security, we use bug reports as our source of information. Bug reports and their associated fix commit messages have been previously used to extract general metrics or infer information about software [17, 21, 23]. Textual features of bug reports have also been used to provide classifications using machine learning techniques [16, 24]. Similar to how background checks, school transcripts, or a history of car accidents are used to determine current reliability aspects of individuals, we use bug reports as a proxy to measure performance and security of a library. Since issue tracking systems such as Github do not provide uniform issue labels (e.g. performance) for bug reports across different repositories, we resort to a different method to extract this information. Specifically, we use a combination of a keyword-based approach and a classifier to label bug reports as performance or security related. For simplicity, we will only refer to Performance as the metric being discussed; however, we use the same exact methodology to extract Security data.

To extract Performance data for our list of libraries, we first collect the titles of all bug reports from Github or JIRA, based on where the issues are tracked for each library. Our performance extraction methodology is then divided into two steps. The first step consists of filtering bug reports based on keywords. The second step relies on automatically classifying the resulting bug reports using a machine learning classifier. We use a two-step methodology, as using only machine learning classifications resulted in a large number of false positives. We detail these steps in the following paragraphs.

Bug Report Filtering. We filter bug reports of a given library by searching the titles of these reports for keywords related to general performance problems. The list of keywords can be found in our public repository. Examples of the keywords used include *memory leak*, *overflow*, and *deadlock*.

Training Dataset. In order to apply machine learning classifiers, we use a training dataset of 1,000 titles of bug reports, 500 labeled as performance bug reports and the other 500 as non-performance. To create this dataset, we first use existing manually classified bug reports from Ohira et al. [22] which contained 320 performance reports and 161 security reports. Additionally, in order to have 500 positive examples, we complement the dataset by adding bug reports that we manually classified. Our manual classification was done as follows: we searched the Bugzilla issue tracking system for bug reports. As subject projects, we used Tomcat, Eclipse, Ant, Thunderbird, and Firefox to perform our search as they have a large number of bug reports and do not overlap with any of our target libraries. We manually classified the titles of bug reports as performance or non-performance and agreed on the labels. We then collected the titles of the bug reports which were classified as performance.

Bug Report Classification. Using the training data set mentioned in the last paragraph, we trained a machine learning classifier to provide performance or non-performance classifications for bug reports, using only the title of the bug report as the input data. We omit bug descriptions from our input since we found that they often consist of steps on how to reproduce an issue, which do not provide valuable information to the classifier, and often add noise. For each bug report title, we eliminate stop words, and stem the remaining words from the resulting text. We then calculate the inverse term frequency of the resulting text and used it as input to a Multinomial Naive Bayes classifier. We use Multinomial Naive Bayes as it produced better f1-score numbers when compared to other classifiers that we tried. Our comparison of classification algorithms can be found in our repository. We validated our classifier using a stratified 10-fold cross-validation and achieve a recall of 79%, a precision of 87%, and a f1-measure of 83% (recall of 89%, precision of 88%, and f1-measure of 88% for our security classifier). For our final performance metric for a given library, we report the percentage of bug reports classified as performance-related out of all issues found in the issue tracking system of the library. For this metric, we consider all issues regardless of state, as closed issues can reveal past problems of a library, while open reports may reveal current problems. Using this methodology, we calculate that mockito had 2.14% performance-related issues and 0% security-related issues in its issue tracking system.

3.6 Last Modification Date.

Definition and Intuition. We refer to the *Last Modification Date* of a library as the last time changes were done to its code repository. While the Release Frequency of a library provides an estimation on how often the software is updated, it does not provide information about the recency of the library. Last Modification Date may indicate whether the development of a library is still active.

Extraction Methodology. We extract the date of the last commit made in the Github repository of the library. The Last Modification Date for mockito was January 15, 2018.

3.7 Last discussed on Stack Overflow

Definition and Intuition. This metric refers to the last time a question was posted about a given library in the popular Q&A website Stack Overflow. A lack of recent questions in Q&A websites may raise some red flags about the activeness of a library's community.

Extraction Methodology. For this metric, we collect the Stack Overflow tags corresponding to the libraries in our list. We identify the tag for each library by using the search by tag functionality on the Stack Overflow website. Using the Stack Overflow API, we then search for the most recent questions containing the identified tag of a given library and extract the date of latest posted question. As an example for this metric, mockito was last discussed on January 15, 2018 at the time we prepared the data for our survey.

4 DEVELOPER SURVEY

To gather data that would help us answer our research questions, we extract the metrics described in Section 3. We use this data to create metric-based comparisons for our survey of Java developers. We now describe the details of this process.

4.1 Surveyed Domains and Libraries

To compare similar libraries, we chose 10 of the most popular Java library domains in MVNRepository [3], a website that categorizes Java libraries based on domains. These domains were *testing*, *database*, *utilities*, *xml processing*, *logging*, *object relational mapping*, *json processing*, *mocking*, *security*, and *cryptography*. For each domain, we investigate the available Java libraries by consulting the same website. Since we need information from software repositories and issue tracking systems, we discard libraries which do not have these systems publicly available. Additionally, we ignore libraries where the main language used in their issue tracking systems is not English, since some of our methodologies rely on text analysis. Based on this selection criteria, we apply our extraction methodology to 34 Java libraries (full list on our artifact page [2]). For each domain, we record the gathered metric data which can then be queried to create a table similar to that shown in Figure 1.

4.2 Survey Design

To host the survey, we built our own website and structured the survey as follows. After answering background questions about their professions and level of proficiency in Java, participants see a list of the available library domains and are asked to choose one domain to evaluate. To have a balanced set of responses across domains, we hid domains that had a high number of evaluations compared to the other domains. This ensured that we received a balanced set of evaluations for each domain. Thus, the list of displayed domains dynamically changed depending on the number of evaluations per domain. Participants then see a metric-based library comparison table for that domain, similar to that in Figure 1. The order of the rows with metric data did not change. Participants are able to hover over each information icon in the table to see an explanation of the presented metric. Participants are then asked to answer a set of domain-specific questions based on the given comparison. Finally, participants are given the choice to either evaluate another domain of libraries, or to proceed to the exit survey, which contains general questions about our metric-based comparison of libraries. The following provides details of the questions in each part of the survey.

4.2.1 Background Questions.

Q_{B1} *What is your current occupation?* undergrad. student, grad. student, academic researcher, industrial researcher, industrial developer, freelance developer, or other.

Q_{B2} *How many years of Java development experience do you have?* <1 year, 1-2, 2-5, 6-10, 11+ years.

4.2.2 Domain-specific questions. These questions are shown to participants after they select a library domain of their choice from a list of available options. The questions appear below the metric-based comparison table of the given domain.

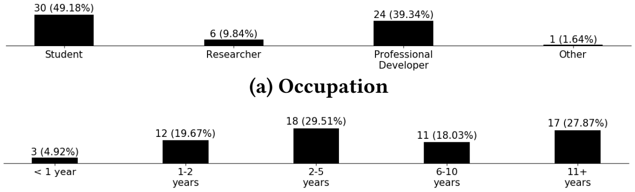
Q_{D1} *Based on the presented information, which of the above libraries would you select if you were looking for a [domain] library to use?* A dropdown list containing the names of the libraries shown in the table. Only 1 library could be selected. The purpose of this question, as well as the next 2, is to ensure that participants think about the presented information.

Q_{D2} *Which of the above libraries have you used before?* A checklist containing the names of the libraries shown in the table. Multiple libraries could be selected.

Q_{D3} *Name any other libraries from this domain that you have used before.* Free-text.

Q_{D4} *Explain your reasons for your choice in Q_{D1}. Which metrics, if any, influenced your decision?* Free-text.

4.2.3 Exit Survey. Once participants finish evaluating one or more domains, they proceed to the exit survey.



(a) Occupation

(b) Years of experience using Java

Figure 2: Background of survey participants.

Q_{E1} *On a scale of 1 to 5, with 1 being not useful at all to 5 being very useful, how would you rate the usefulness of the following metrics?* A list of the 9 metrics is presented, with a Likert scale from 1 to 5 for each metric.

Q_{E2} *On a scale of 1 to 5, with 1 being not useful at all to 5 being very useful, how useful do you find the above metric-based comparison for selecting a library to use from a given domain?* Likert scale from 1 to 5.

Q_{E3} *Are there additional metrics that you think might be helpful for comparing libraries in a given domain?* Free-text.

Q_{E4} *Please let us know if you have any further comments about the above metric-based comparison of libraries.* Free-text.

4.3 Participant Recruitment

Our survey is designed to uncover metrics that influence the decisions of software developers when choosing Java libraries to use. Our target audience is Java developers in general. To recruit different kinds of participants, we employed three recruitment strategies.

4.3.1 Github Recruitment. To recruit developers who have used one of our subject libraries before, we search for Github users who contributed code to client projects of our subject libraries. As it is not possible to contact Github users directly, we gathered e-mail addresses as follows. We use the Github API to search for Java repositories that included code that references API elements of our subject libraries as the only filtering criteria. From these results, we look at the files containing our searched API elements and collect the git commits associated with them. Finally, we discard git commits that were older than 6 months and obtain the e-mail addresses of the authors of the remaining commits. We collected a total of 298 e-mail addresses.

4.3.2 Stack Overflow Recruitment. Our second recruitment strategy consisted of recruiting Stack Overflow users who have been involved in discussions of two subject libraries of the same domain. Such a discussion indicates that they were previously shopping for libraries and thus getting the input of such users is important. Since Stack Overflow does not offer functionality to contact users directly, we search for questions containing at least two tags of our subject libraries and collected e-mail addresses by visiting the personal websites listed in the profiles of the users involved in those questions. Given the manual nature of this methodology and the fact that not all users have websites on their profiles, we collected only a total of 20 e-mail addresses in this step.

4.3.3 Snowball Sampling. Finally, our third strategy consisted of snowball sampling [10] where we simply asked any Java developer to fill out the survey. We sent email invitations to the undergraduate and graduate mailing lists of the Computer Science department at

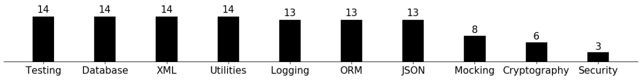


Figure 3: Number of evaluations for each domain

our university, who may have forwarded the email to others. We also advertised the survey on social media accounts and invited others to promote the survey.

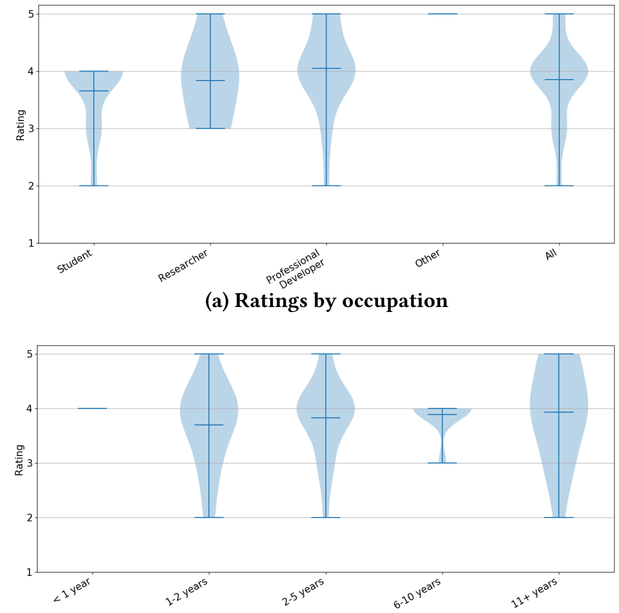
5 RESULTS

We received a total of 61 responses in our survey. From the 318 e-mails that we sent to Github and Stack Overflow users, we collected 24 responses (7.5% response rate), with 21 participants recruited from Github and 3 from Stack Overflow. The remaining 37 responses were obtained with the snowball sampling recruitment. Out of the 61 total responses, 53 participants completed all sections of the survey, while 8 participants answered only the background and domain-specific questions but did not answer the exit questions. A total of 22 participants evaluated more than 1 domain, while the rest assessed only 1. We first give a detailed breakdown of the collected survey data and then proceed to answer our 3 research questions. All our reported statistics use a pairwise Wilcoxon sum rank test to observe any statistically significant differences between metric ratings, using $\alpha = 0.05$. Since we perform multiple tests, we use the Holm's adjustment method for our p values. To estimate effect sizes of any significant differences, we use Cliff's delta with the following ranges [11, 17]: small for d < 0.33, medium for 0.33 <= d < 0.474 and large for d >= 0.474. We use violin plots to show the density of the data at different values.

5.1 Survey Data Breakdown

5.1.1 Background of Participants. Figure 2a shows the distribution of participant's backgrounds. For simplicity and better visualization, undergraduate and graduate students are grouped under *Student*. Similarly, we use the category *Researcher* for both industrial researcher and academic researcher, and *Professional Developer* for both industrial developer and freelance developer. Students present the highest percentage of participants (49.18%), followed by Professional Developers (39.34%). Note that there was one participant who picked the other category in occupation and indicated that they are a business analyst. Figure 2b shows the distribution of years of Java experience among participants. Three quarters of the participants (75.41%) had at least 2 years of Java experience.

5.1.2 Domain & Library Breakdown. The number of evaluations per library domain are shown in Figure 3. Note that the same participant may have evaluated more than one domain. Testing, Database, XML, and Utilities are the most evaluated domains, each consisting of 14 domain-specific survey responses, while Security is the least evaluated with 3 responses. In 29.46% of responses to Section 4.2.2, participants chose a library they had not previously used. We provide a detailed breakdown of the selected libraries in our artifact page.



(a) Ratings by occupation

(b) Ratings by experience

Figure 4: Usefulness ratings for Q_{E2}

5.2 RQ1. Is a metric-based comparison of libraries useful for developers when selecting a software library to use?

We start by first answering RQ1, which evaluates the overall usefulness of metric-based library comparisons. We do so by analyzing the 53 answers for Q_{E2} of the survey. Figure 4a shows the distribution of ratings for Q_{E2}. The first four violin plots in the figure show the distributions of ratings per occupation group, while the fifth plot shows the distribution of ratings by all participants. When considering all participants, the right-most plot in Figure 4a shows that the highest frequency of answers is concentrated near rating 4 (i.e., *Useful*) based on the Likert scale. The mean rating of all participants for Q_{E2} is 3.85 and the median is 4.

We now look at the ratings per participant group, but ignore the other category since it has only one participant. Looking at the first three plots in the figure, we can see that Professional Developers have rated the usefulness of a metric-based comparison the highest with a mean rating of 4.05, Researchers have an average rating of 3.83, and Students have an average rating of 3.65. Despite of this, we find no statistically significant differences between the ratings of the occupations. While Figure 4b shows a small ascending trend of the ratings as experience increases starting from the 1-2 years group, we also find no significant differences between the ratings of the Java experience groups. We can therefore conclude that most participants from all backgrounds found our metric-based comparison useful for selecting libraries.

Finding.1: When comparing software libraries, our participants, regardless of background, find a metric-based comparison of libraries useful (mean rating = 3.85).

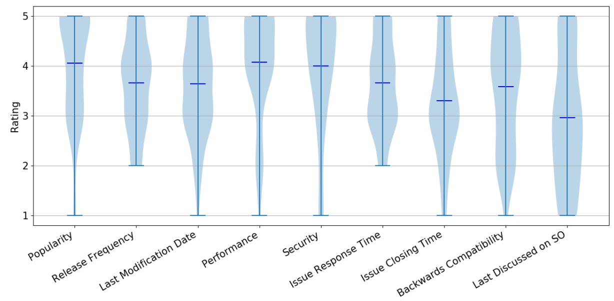


Figure 5: Rating distributions of Q_{E1} for all metrics.

5.3 RQ2: Which metrics influence developers' library selections when comparing software libraries?

To answer the second research question, we use three sources of data from our survey. The first is the individual ratings of each metric in question Q_{E1}. The second is the free-form answers to Q_{D4} where participants explicitly mentioned which metrics affected their decision when selecting a library and why. The third is the free-form question Q_{E3}, where participants mention additional metrics they would like to see.

Figure 5 shows the distribution of the ratings for each metric and also indicates the mean score. All metrics have a mean above 3.0 with the exception of the *Last Discussed On Stack Overflow* metric, whose mean is 2.96. *Performance*, *Popularity*, and *Security* were the 3 highest rated metrics by participants, with mean ratings 4.08, 4.06, and 4.00 respectively. Additionally, we find that each of these 3 metrics has statistically significant differences with the metrics *Issue Closing Time* and *Last Discussed on Stack Overflow*. Analyzing the magnitude of these differences, we observe medium effect sizes (d = 0.38) between Popularity and Issue Closing Time, and between Popularity and Last Discussed on Stack Overflow (d = 0.45). For *Security*, the results show a medium effect size with Issue Closing Time and Last Discussed on Stack Overflow (d = 0.38 and 0.42 respectively). Finally, for Performance, the tests also reveal medium effect sizes with Issue Closing Time (d = 0.41) and with Last Discussed on Stack Overflow (d = 0.45). Thus, we can conclude that compared to the other metrics, Performance, Popularity and Security are indeed metrics that have more influence on developers' decisions.

To gain insights about the top-rated metrics, we analyze the free-form answers from Q_{D4}. We use P1 to PN when quoting different participants. We observe that participants are concerned about possible unnecessary overhead to their applications that libraries may add, which might serve as explanation for the high mean for Performance. For example, P1 said that *"[...] logging already added some overhead, I don't really want any performance issues with the library that I'm using"*. For Popularity, we observe that participants associate this metric with library quality and support, as the comments of P2, P3, and P4 suggest: *"Popularity is a good proxy for quality"*, *"... These 2 metrics [Popularity and Last Modification Date] are a good indicator of the quality of a library"*, and *"It's popular so I guess it should have something good"*. Moreover, Popularity is also associated with library support, as we can infer from these comments by P4, P5, and P6: *"Popular libraries tend to have the most support"*, *"...usually more popular libraries are better developed and supported"*, and *"Popularity reflects the sustainability of the library"*. Although Security is positioned as the third highest rated metric, we did not find explanations for this fact in our collected answers. However, since several of our subject domains deal with handling of data, we speculate that developers most probably consider integrity and confidentiality of data as an important factor when choosing libraries.

Finding.2: Performance, *Popularity* and *Security* are rated highest, and each had two statistically significant differences in their ratings w.r.t other metrics.

To find additional metrics that developers care about that we did not consider in our comparison, we analyze the free-form answers for Q_{E3} of the Exit Survey. This question explicitly asked participants about additional metrics they would like to see. In total, 39 of our participants left comments for this question. To analyze their responses, we use an open coding approach from grounded theory [5], specifically card sorting. The two authors of this paper wrote each metric that a participant mentioned on a piece of paper. Then we iteratively grouped related metrics, until distinct categories were formed. Table 1 shows the categories we found, along with the number of associated answers and the definition of the category. Additionally, we did not understand the comments by 4 participants so we do not include them in our categorization. In Section 6, we discuss the implications of our findings and how existing research can be used to extract some of the additional metrics mentioned by participants.

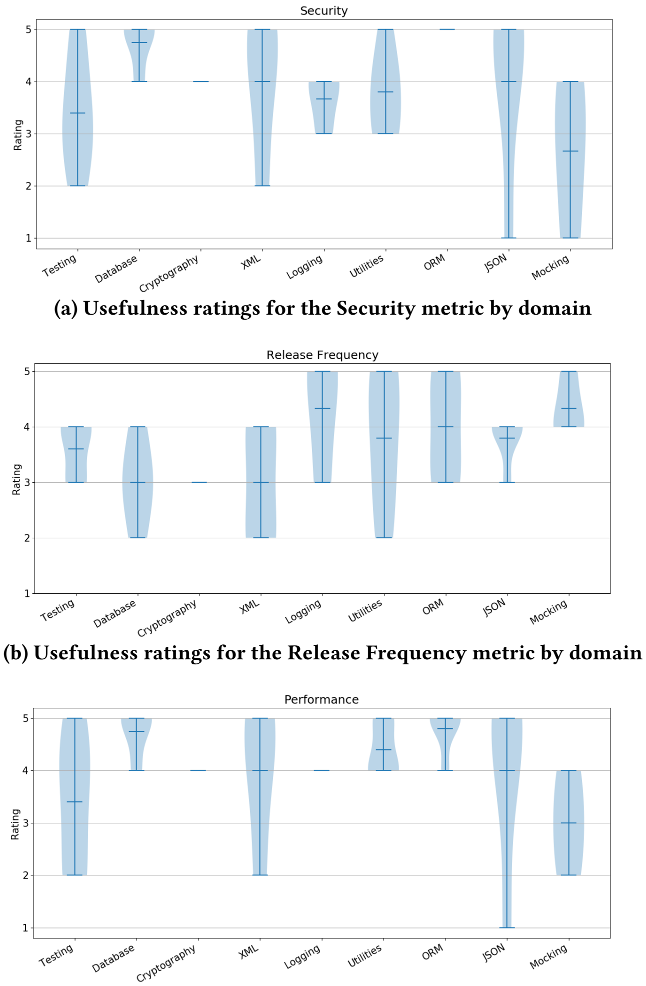
Finding.3: Additional metrics related to documentation and usability of a library are highly desired by developers.

5.4 RQ3. Does the library domain affect metric usefulness?

Since our metric comparison is organized by domain, we are interested to see if the perceived usefulness of the metrics depends on the domain. To investigate this, we use two sources of data. The first is the metric ratings from Q_{E1} from the exit survey, and the second is the free-form answers to Q_{D4} from the domain-specific questions where participants explicitly mention the metrics that affected their decision.

For Q_{E1}, we analyze the metric ratings per domain by using survey responses that evaluated a single domain. This means that we do not consider participants who evaluated more than one domain, such that we can get a one-to-one mapping between domain and metric rating. We have 35 such ratings to analyze. We find that while there are no large differences among the average ratings per domain for metrics such as Popularity, Release Frequency, and Last Modification Date, the Security metric shows large differences between distinct domains. We visualize three metrics, including Security, that show large differences among domains in Figure 6. Plots for the remaining metrics are on our artifact page. Note that we do not include a column for the Security domain in our plots since there are no responses that evaluated only this domain.

Given that the number of one-to-one domain-metric ratings are not that high (e.g., 1 rating in the Cryptography domain vs. 5 ratings in the JSON domain), we do not perform statistical tests to



(a) Usefulness ratings for the Security metric by domain

(b) Usefulness ratings for the Release Frequency metric by domain

(c) Usefulness ratings for the Performance metric by domain

Figure 6: Examples of differences in metrics by domain

compare these populations since the results of the tests would be meaningless. Instead, we triangulate the descriptive statistics in Figure 6 with the answers to Q_{D4}, where participants discussed which metrics influenced their decision to select a library. We have 108 comments in Q_{D4}, since many participants evaluated more than one domain. We analyze the comments and count the number of times each metric is mentioned. We present a heat map in Figure 7 based on these counts. Darker colors indicate more mentions by participants. The heat map confirms our results from the metric ratings in the previous paragraph. We can see that Popularity is frequently explicitly mentioned as a reason to choose a library for most domains. In contrast, the Last Discussed on Stack Overflow metric was barely mentioned by participants. As previously discussed, Security seems to be a metric that is only relevant in some domains, such as databases and cryptography, but completely irrelevant in other domains such as utilities or testing. To understand why metric usefulness may vary among domains, we look at explanations provided by participants in Q_{D4} as well as any relevant comments from the general additional comments part Q_{E4}.

Table 1: Categories of additional metrics and the number of participants mentioning these metrics.

Category	# of participants	Definition
Documentation Quality	14	Recency and availability of library documentation and learning materials
Library Usability	8	Metrics related to the ease of use and learning curve of a library.
General Library Information	8	Statistics about a library and its public repository.
Library Functionality	5	Information about the functionality offered by a library.
Community Support	5	Statistics about community aspects from a library.
Legal	3	Information about licensing and ownership of a library
Compatibility	3	Information about the library's compatibility with other software such as libraries, platforms, and programming language versions.
Dependencies	2	Other software that needs to be installed to use a library.
Library Alternatives	2	Information about similar libraries including from other programming languages.
Crowd-sourced Opinions	2	Opinions and reviews found in Q&A websites
Robustness	2	Reliability of a library.
Quality Assurance	3	Information about testing suites and continuous integration statistics of a library.
Memory	1	Information about the memory usage of a library.

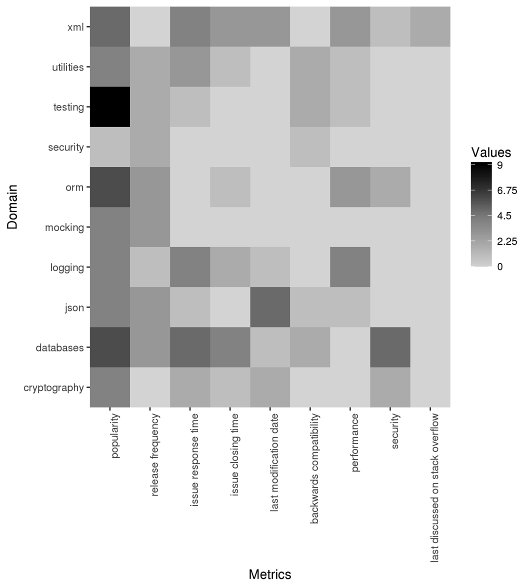


Figure 7: Heatmap of frequency of metrics mentioned by participants in Q_{D4}. Darker colors indicate higher frequency.

Security is intrinsic to certain domains: *"Since this is a crypto API, I'd like to use a library that has the least security issues"* - P1. On the other hand, Security provides little use for specific domains to some participants, as P7 suggests: *"Importan[ce] of metrics depends on the library. I don't care about the security of JUnit."* We can observe this opinion reflected in Figure 6a, where the testing and mocking domains have low ratings for the Security metric. Similarly, Performance is seen as crucial for some domains, as P8 describes *"In ORM, the first thing I care about is performance in enterprise projects"*. For Release Frequency, shown in Figure 6b, certain domains may provide more stable library releases and therefore, the metric is less important, as P7 points out: *"Testing frameworks do not need to be released all the time. They are stable."*

Finding.4: Some metrics are more intrinsic to certain domains than others. Examples include Security, Performance, and Release Frequency.

6 DISCUSSION

A metric-based comparison of software libraries has two main challenges: (1) knowing which metrics to include and (2) designing a method to accurately extract these metrics and present them in an intuitive way. Our survey focuses on the first challenge: it evaluates whether a metric-based comparison of software libraries is useful to developers, and helps uncover which metrics developers care about when making a choice. However, the comments provided by our participants give us valuable suggestions and insights on how to address the second challenge. In the rest of this section, we discuss improvements and the most requested missing metrics as mentioned by our survey participants and we provide our insights on how these suggested metrics could be extracted.

Table 1 shows that documentation quality is by far the most demanded metric, even though it is hard to quantify, as mentioned by one participant. There is a vast amount of work on how to improve documentation (e.g., [28]), beyond what we can mention in this paper. While concretely quantifying documentation is hard, techniques such as mining various information sources and presenting related links on our metric-based website, e.g. linking the API used in the documentation to the library [29], are feasible. Other options include automating documentation search techniques, such as those presented by Parnin and Treude [25], or using patterns of knowledge in API documentation as a means to assess their quality [18]. We plan to investigate how existing documentation techniques can be summarized and quantified in a way that quickly gives developers an indication of availability and quality.

The next most demanded metric relates to library usability; specifically, how easy it would be to use the API. Since this is also hard to quantify, an option is to leverage the vast amount of research that is able to mine API usage examples [27] and present code snippets to developers, which could then assess if the code look easy to use or not.

Participants also suggested providing general information about a library. For example, surprisingly, two participants mention code size while we thought client developers would not necessarily care about low-level details of the library. Others mention more high-level general information metrics such as absolute number of releases and how old the library is.

Five participants mention that understanding the functionality offered by the library is important, while one participant wishes to see the overlap in functionality with other libraries. Another comment suggests having code examples for each of the library functionalities. The community support around a library seems to also be very relevant for users. While we had metrics related to this aspect, additional metrics such as the size of the community and how often the library is discussed in various blogs and resources were suggested. Such metrics could be automated by mining the web and looking at networks of related developers.

Interestingly, one aspect that we did not consider but that is very important in practice is the licensing of a library. GitHub already states the licenses used in each project, or license mining can also be used [34]. Similarly, it makes sense that developers are interested if a given library is compatible with libraries that they are using, as well as what dependencies they would need to include for this library to work correctly. Techniques for mining library dependencies can also be used to extract such information. Among the remaining categories, crowd-sourced opinions could be obtained from Q&A websites by mining opinions [33].

As suggestions on the current metrics, participants mention improving the presentation of the metric information, such as adding graphical representations of the data (e.g. using trend graphs to visualize metrics such as Issue Response Time) or aggregated scores for each library (e.g. 5-star rating for each library summarizing all of its metrics data.). Finally, based on Finding 4, we believe that displaying customizable metric-based comparisons based on the domain (e.g. displaying the most important metrics of the domain first) may be useful to users. The fact that distinct participants in our survey often rated the same metric differently supports our reasoning that developers may care about different characteristics of a library depending on their needs. This further sustains our belief that it is important to provide developers with all relevant data, in an easily comparable form, and leave them to decide which parts of this data they will use for their choices.

7 THREATS TO VALIDITY

7.1 Internal Validity

Metrics. The list of metrics used in the survey is not comprehensive for the purposes of library comparisons, as Section 6 suggests, and there is room for improvement in terms of our extraction and presentation methods. However, the goal of this is to assess the general usefulness of metric-based comparisons and to gather information and feedback about the metrics that are most important to developers, with the objective of creating a publicly available implementation in the future. Therefore, our survey findings are not affected by potential small inaccuracies of the metrics we employed, especially in a comparison context.

Implementation of scripts. To mitigate any potential bugs in the scripts we use to extract data for our metrics, we manually verified various samples of the results. Additionally, we make our scripts publicly available for others to verify or replicate our work.

Usage of import statements and Boa. Relying on `import` statements to collect popularity information may not reflect actual usage of the imported API elements. However, it provides an upper bound estimation of the popularity of a library. Another threat concerns the age of the dataset of projects used to obtain our data, as it dates to September 2015. While this dataset may not reflect current library usage trends, and would be older than the data used for the other metrics, it still shows the relative popularity since the same data set is used for all libraries and this is sufficient for the purposes of our survey. For our website implementation, we plan to not depend on Boa for this task in order to provide updated data.

Training Dataset for Classification. To create a dataset of bug reports related to performance and security problems, we manually classify bug reports based on their title. This could impact the predictions of the classifier. To mitigate this threat, 2 authors of this paper agreed on the manual classifications of bug reports. Additionally, we use the titles of bug reports provided by other researchers [22]. Performance and Security are difficult metrics to accurately quantify and we plan to investigate alternative methods.

7.2 External Validity

Survey Results. The findings presented in this work are based only on our sample set of participants and may not generalize beyond this context. However, to reduce possible opinion biases, we recruited participants with varying backgrounds and through different sources.

List of Libraries. Our survey website did not provide an exhaustive list of libraries per domain as not all libraries had publicly available repositories and issue tracking systems. This may affect our findings and conclusions, but we believe this effect would be confined to the library choice in Q_{D1}, which was not explicitly used in our data analysis.

8 CONCLUSIONS

Selecting the best-suited software library to use is often a non-trivial task for developers. In this paper, we implemented and evaluated the idea of using data analytics of various software repositories to create metric-based comparisons of libraries to assist developers with this selection. Through the answers of 61 surveyed developers, we found that metric-based comparisons are useful to most developers. Additionally, our survey results showed that Performance, Popularity, and Security are the most useful metrics to developers, with Popularity being useful across all library domains. Finally, participants indicated that they would like to see metrics related to aspects such as usability, documentation, functionality, community support, and licensing. Participants also provided us with some suggestions for presentation improvements of our current metrics. We are currently building a public metric-based library comparison website for the community to use, and plan to take participants' recommendations into account.

REFERENCES

[1] 2018. Git Tags. (2018). https://git-scm.com/book/en/v2/Git-Basics-Tagging.

[2] 2018. Library Metric Comparison Online Artifact. (2018). https://github.com/ualberta-smr/LibraryMetricScripts.

[3] 2018. MVNRepository. (2018). https://mvnrepository.com/.

[4] H. Borges, A. C. Hora, and M. T. Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. *CoRR* abs/1606.04984 (2016).

[5] Juliet Corbin and Anselm Strauss. 2008. *Basics of qualitative research. Techniques and procedures for developing grounded theory* (3rd ed.).

[6] Daniel Alencar da Costa, Shane McIntosh, Uirá Kulesza, and Ahmed E. Hassan. 2016. The impact of switching to a rapid release cycle on integration delay of addressed Issues: an empirical study of the Mozilla Firefox project. In *Proc. of the International Conference on Mining Software Repositories (MSR)*. 374–385.

[7] Fernando López de la Mora and Sarah Nadi. 2018. Which library should I use? A metric-based comparison of software libraries. In *Proceedings of the 40th International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE NIER '18)*.

[8] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 422–431.

[9] Emanuel Giger, Martin Pinzger, and Harald Gall. 2010. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE '10)*. 52–56.

[10] Leo A. Goodman. 1961. Snowball Sampling. *Ann. Math. Statist.* 32, 1 (03 1961), 148–170.

[11] Robert J Grissom and John J Kim. 2005. *Effect sizes for research : a broad practical approach.* Mahwah, N.J. ; London : Lawrence Erlbaum Associates. Formerly CIP.

[12] Emitza Guzman, David Azócar, and Yang Li. 2014. Sentiment analysis of commit comments in GitHub: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. 352–355.

[13] Andre Hora and Marco Tulio Valente. 2015. Apiwave: keeping track of API popularity and migration. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME '15)*. 321–323.

[14] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging library migrations: a case study for the Apache Software Foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. 154–164.

[15] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. 2012. Do faster releases improve software quality?: an empirical case study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12)*. 179–188.

[16] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. 2010. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 1–10.

[17] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. 477–487.

[18] Walid Maalej and Martin P Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[19] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops (IWPSE-Evol '09)*. 57–62.

[20] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. 2010. Mining API popularity. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques (TAIC PART'10)*. 173–180.

[21] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. 215–225.

[22] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto. 2015. A dataset of high impact bugs: manually-classified issue reports. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. 518–521.

[23] Marco Ortu, Bram Adams, Giuseppe Destefanis, Parastou Tourani, Michele Marchesi, and Roberto Tonelli. 2015. Are bullies more productive?: empirical study of affectiveness vs. issue fixing time. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. 303–313.

[24] Nitish Pandey, Abir Hudait, Debarshi Kumar Sanyal, and Amitava Sen. 2018. *Automated classification of issue reports from a software issue tracker.* 423–430.

[25] Chris Parnin and Christoph Treude. 2011. Measuring API documentation on the web. In *Proceedings of the 2Nd International Workshop on Web 2.0 for Software Engineering (Web2SE '11)*. ACM, 25–30.

[26] M. M. Rahman, C. K. Roy, and D. Lo. 2016. RACK: automatic API recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 349–359.

[27] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (May 2013), 613–637.

[28] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, and others. 2017. On-demand developer documentation. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 479–483.

[29] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 643–652.

[30] C. Teyton, J. R. Falleri, and X. Blanc. 2012. Mining library migration graphs. In *2012 19th Working Conference on Reverse Engineering*. 289–298.

[31] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A study of library migrations in Java. *Journal of Software: Evolution and Process* 26, 11 (Nov. 2014), 1030–1052.

[32] Ferdian Thung. 2016. API recommendation system for software development. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 896–899.

[33] Gias Uddin and Foutse Khomh. 2017. Automatic summarization of API reviews. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*.

[34] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk. 2017. Machine learning-based detection of open source license exceptions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 118–129.

[35] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: a large scale study. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. 138–147.