Journal of Software Engineering Research and Development, 2021, 9:1, doi: 10.5753/jserd.2021.548

Mining Experts from Source Code Analysis: An Empirical Evaluation

Johnatan Oliveira [**Federal University of Minas Gerais (UFMG)** | ***johnatan.si@dcc.ufmg.br***]
Markos Viggiato [**University of Alberta** | ***viggiato@ualberta.ca***]
Denis Pinheiro [**Federal University of Minas Gerais (UFMG)** | ***denisppinheiro@gmail.com***]
Eduardo Figueiredo [**Federal University of Minas Gerais (UFMG)** | ***figueiredo@dcc.ufmg.br***]

Abstract Modern software development increasingly depends on third-party libraries to boost productivity and quality. This development is complex and requires specialists with knowledge in several technologies, such as the nowadays libraries. Such complexity turns it extremely challenging to deliver quality software, given the pressure. For this purpose, it is necessary to identify and hire qualified developers, to obtain a good team, both in open source and proprietary systems. For these reasons, enterprise and open source projects try to build teams composed of highly skilled developers in specific libraries. However, their identification may not be trivial. Despite this fact, we still lack procedures to assess developers skills in widely popular libraries. In this paper, we first argue that source code activities can identify software developers' hard skills, such as library expertise. We then evaluate a mining-based strategy to reduce the search space to identify library experts. To achieve our goal, we selected the 9 most popular Java libraries and 6 libraries for microservices (i.e., 15 libraries in total). We assessed the skills of more than 1.5 million developers in these libraries by analyzing their commits in more than 17 K Java projects on GitHub. We evaluated the results by applying two surveys with 158 developers. First, with 137 library expert candidates, they observed 63% precision for popular Java libraries' used strategy. Second, we observe a precision of at least 71% for 21 library experts in microservices. These low precision values suggest space for further improvements in the evaluated strategy.

Keywords: *Library Experts, Software Skills, Expert Identification, Mining Software Repositories.*

1 Introduction

Software development has become increasingly complex, both in open-source and proprietary systems (Damasiotis et al., 2017). Such complexity makes it extremely challenging to deliver software with quality in time and may hinder developers' participation in worldwide repositories of source code, such as GitHub (Viggiato et al., 2019). To contribute to open-source projects or hire developers (in the case of a company), identifying the developer with the right skills for a good team is a hard task (Garcia et al., 2007; McCuller, 2012). Besides, in many cases, project managers must build teams of skilled developers in relevant libraries. However, decisions made during the hiring process are a well-known decisive factor to the success of a software project (Tsui et al., 2016). Providing a more reliable way of identifying developers' skills can help project managers make the right decision when hiring or attracting the right developers for an open-source project. The task of finding experts in specific technologies is especially complex, despite the existence of business-oriented social networks, such as LinkedIn, where developers write about their attributes and qualifications. This type of platform is commonly used for the online recruitment of professionals. However, the reliability and accuracy of the information provided in such media are not guaranteed (Brown and Vaughn, 2011). For instance, some individuals can overvalue their skills and omit some skills in a self-authored curriculum.

The most commonly used strategies to find experts have their limitations (Tsui et al., 2016; Constantinou and Kapitsaki, 2016). For instance, the analysis of the curriculum from LinkedIn or in paper format can omit desirable skills. Besides, developers may have difficulty to express their qualifications (Tsui et al., 2016). Sometimes, the developer has a specific ability, but s/he considers it irrelevant. In another situation, the developer cites many skills but does not have expertise in the technologies mentioned (Constantinou and Kapitsaki, 2016). Even large companies may rely on curriculum analysis, and this type of research may have inaccurate or outdated information. Besides, even talent recruiters may incorrectly identify the developer skills or identify other skills that are not the organization's focus. Hiring lowly skilled software developers can lead to additional costs, efforts, and resources for training them, or expending more time and resources hiring others (Constantinou and Kapitsaki, 2016; Sommerville, 2015). However, these costs can be reduced if companies identify with more precision best developers according to a job opening.

Several software developers have used social coding platforms, such as GitHub and BitBucket, to showcase their work, hoping that this may help them be hired for a better job. Developers use these social coding platforms to demonstrate their skills and create an online profile about their projects (Constantinou and Kapitsaki, 2016). Some contributors are even using these platforms' social aspects to infer project popularity trends and promote themselves more efficiently through specific projects and collaborations in other open-source projects (**?**). In some cases, profiles derived from accounts of social platforms, such as GitHub, are considered even more reliable than a curriculum from LinkedIn, concerning the technical qualifications of a job candidate (Constantinou and Kapitsaki, 2016). Therefore,

data exploitation from coding platforms is a promising way for potential employers to identify and assess several candidates in real situations (Capiluppi et al., 2013).

GitHub has been widely used in several works mainly because it provides several user-based summary statistics, such as the number of contributions in the last year, the number of forked projects, and the number of followers. For instance, some works have used this platform to identify appropriate developers for cross-project bugs (Ma et al., 2017), identification of reuse opportunities (Oliveira et al., 2016) and collaborations between projects (Dabbish et al., 2012). Different approaches have been used to investigate the skills of developers from GitHub (Saxena and Pedanekar, 2017; Mockus and Herbsleb, 2002; Greene and Fischer, 2016). For instance, prior work conducted interviews with members of GitHub to understand the hiring process (Marlow and Dabbish, 2013). We did not compare the results with other approaches because our strategy is very different from the others. Therefore, our strategy complements related work by automatically reducing the search space to support library experts' identification. This paper is an extension of our previous work (Oliveira et al., 2019) that proposed and evaluated a strategy to identify library experts from source code, named JExpert. Our main goal is to reduce the search space to identify library experts. We list the following new contributions to this submission compared to the original paper.

1. We present and analyze data of all identified expert candidates by means of new boxplot charts.
2. We include a novel classification and discussion of experts in four categories.
3. We include additional analysis of the library experts by proposing a novel heuristic to rank the top experts of each library.
4. We perform a new identification of experts in microservices libraries.
5. We conducted an additional survey to calculate the strategy precision on identifying experts in microservices.
6. We include additional discussion about the negative results of the evaluated metrics.

In this paper, we evaluate the feasibility of identifying software developers' hard skills; that is, library expertise from source code analysis. We rely on GitHub data to support the identification of the skills of developers based on their contributions. From each type of developer contribution, we aim to identify essential developers skills and evaluate the applicability and precision of the strategy. In the applicability evaluation, we performed a mining study with the top-9 most popular Java libraries from GitHub, aiming to identify library experts in these libraries. In total, we analyzed more than 16 thousand projects and 1.5 million developers. In the precision evaluation, we designed and sent a survey to more than 1 thousand developers identified for these libraries. We received answers from 158 developers. As a result, we observe that it is possible to reduce the search space to identify experts from source code. We also note that the strategy provides meaningful information to recruiters, such as the history of written lines of code (LOC) for each library. These details about the developers can improve the selection of candidates.

Our key contributions are threefold:

- we empirically evaluate the applicability and precision of identifying library experts from source code analysis. In addition, we propose a tool to support the strategy;
- we identify 1,045 experts in top-9 Java libraries with a precision of about 63%;
- we identify 136 experts from microservices libraries with a precision of about 71%.

Low precision values indicate space for future research in this subject. The remainder of this paper is organized as follows. In Section 2, we describe our analysis by detailing the strategy to identify library experts, dataset, and our research questions. Section 3 presents the results of the applicability evaluation to identify library experts. Section 4 shows the results to survey with top-9 library experts. Section 5 shows the results concerning a survey with library experts in microservices. Section 6 shows details about a tool developed to support the strategy. Section 7 presents and discusses threats to validity. Related work is discussed in Section 8. Finally, Section 9 discusses the concluding remarks and future work.

2 Study Settings

This section describes the protocol to evaluate the identification of library experts through an empirical study. Section 2.1 presents the aims of our study and the research questions we address. Section 2.2 shows the steps performed to evaluate the expert candidates. Section 2.3 describes the used dataset.

2.1 Goal and Research Questions

This study's primary goal is to evaluate the applicability and precision of a strategy to reduce the search space to identify library experts from source code analysis using software repositories. We are interested in whether the strategy can significantly reduce the search space to identify experts in a specific library. We are also concerned with assessing the relevance of the results provided by the strategy. For this purpose, we select the 10 most popular and standard Java libraries among GitHub developers. We also selected 6 popular libraries for microservices. One library was later excluded (Section 2.3). Therefore, we evaluate the strategy with the 9 most popular Java libraries and 6 libraries of microservices. To achieve this goal, we use the Goal-Question-Metric method to select measurements of source code. The GQM method proposes a top-down approach to defining measurement; goals lead to questions that are then answered with metrics (Basili et al., 1994).

Table 1 shows the GQM with the research questions and metrics investigated in this study. As mentioned, the goal of this paper is to reduce the search space to identify library experts from source code. Therefore, from this goal, we check if it is feasible to analyze the source code to identify library experts. Through RQ1, we are interested in investigating the efficiency of the number of commits (metric) to indicate the level of activity of a developer in a specific library. In other words, we aim to analyze the number of commits involving

a specific library performed by a developer to compute their activity level in the library.

With RQ2, we aim at assessing the knowledge extension based on the number of imports to a specific library. From all imports made by a developer at the source code, we investigate the number related to the particular library. Finally, the last research question (RQ3) analyzes the knowledge intensity of the developers from the number of LOC related to the library (metric). In this last question, we aim to evaluate the amount of LOC implemented by a developer using a specific library. For this purpose, we evaluate the relation of total LOC and LOC related to a particular library.

Table 1. The Metrics Analysis as GQM method

Questions	Metrics
RQ1– How to evaluate the level of activity of a developer in a library?	Number of commits
RQ2– How to evaluate the knowledge extension of a developer in a library?	Number of imports
RQ3– How to evaluate the knowledge intensity of a developer in a library?	Lines of Code

2.2 Evaluation Steps

This section describes the steps to evaluate the identification of library experts from source code. To answer the research questions presented in Section 2.1, we designed a mixed-method study composed of four steps: 1) *Library Selection*, 2) *Dataset Collection*, 3) *Expert Identification*, and 4) *Survey Application*. Figure 1 presents the steps of our research, which are discussed next. For *Library Selection* (Section 2.3), we selected the top-10 most popular libraries in the Java programming language to identify library experts. We also selected 6 libraries for microservices to favor external validity. In the *Dataset Collection* step (Section 2.3), we clone the projects that contain these libraries from GitHub. For *Identification of Library Experts* (Section 3.1), we compute the skills of developers based on three metrics: *Number of Commits*, *Number of Imports*, and *Lines of Code*. These metrics are presented in Section 3.1. Finally, we performed two *survey studies*. These surveys were conducted to evaluate the precision of the strategy according to the responses of developers. Section 4.1 and 5.1 present details about the surveys.

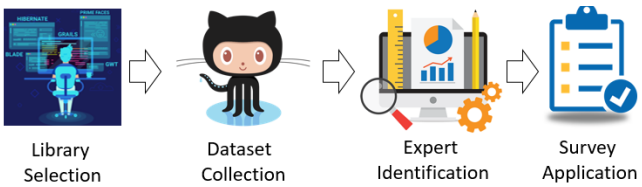

Figure 1. Study Steps

2.3 Dataset

To create our dataset, we select the 10 most popular and common Java libraries among GitHub developers: Hibernate, Selenium, Hadoop, Spark, Struts, GWT, Vaadin, Primefaces, Apache Wicket, and JavaServer Faces. This selection was made based on a survey provided by Stack Overflow[1] in 2018 with answers of over 100,000 developers around the world. Table 2 summarizes the definitions of each library (top-10). All definitions of the libraries were retrieved from Stack Overflow and their Web pages. We selected Java because it is one of the most popular programming languages[2] and there are many Java projects available on GitHub.

Microservices have become most popular in the last years, together with the spread of DevOps practices (Pahl, 2015). We can see a significant increase in the use of microservices architectural style since 2014 (Klock et al., 2017), which can be verified in the service-oriented software industry where the usage of microservices has been far superior when compared to other software architecture models (Alshuqayran et al., 2016). Furthermore, a microservice usually runs on its own process and communicates using standardized interfaces. In practice, microservices are widely used by large Web companies, such as Netflix and Amazon (Alshuqayran et al., 2016). For these reasons, we aim to identify experts of microservices in 6 libraries: Apache Karaf, Apache Spark, JavaEE, Netflix, Spring Boot, and Swagger. Table 3 also summarizes the definitions of each library, but now concerning microservices. These definitions were retrieved from Stack Overflow and their Web pages.

Figure 2 illustrates the criteria for defining our dataset. To achieve more realistic results for software development, we apply the following exclusion criteria. (1) We excluded systems with less than 1 KLOC because we considered them toy examples or early-stage software projects. (2) We removed projects with no commit in the last 3 years because the developers may forget their code (Krüger et al., 2018). Finally, in the last exclusion criteria, (3) we removed projects which did not contain imports related to the selected libraries. Besides, we excluded all official projects of these libraries because we assume all library project developers are experts in the corresponding library. In popular Java libraries, we also removed libraries with less than 100 projects (e.g., JavaServer Faces). We need a representative number of projects to evaluate our strategy. We analyze only files with extension .java. The same process was made to projects of libraries of microservices. Therefore, we end up analyzing 15 libraries in this study.

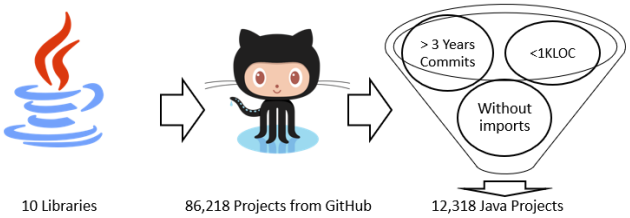

Figure 2. Steps for Collecting Software Projects from GitHub

Table 4 shows the number of remained projects after each step in our filtering process. The first part of this table shows the results for top-10 Java libraries, and the second part

[1]https://insights.stackoverflow.com/survey/2018#most-popular-technologies

[2]https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

Table 2. Library Descriptions

Library	Description
Hibernate	Hibernate is a library of object-relational mapping to object-oriented.
Selenium	A test suite specifically for automating Web.
Hadoop	A library that facilitates the use of the network from many computers to solve problems involving massive amounts of data (Tong et al., 2016; Ye, 2017).
Spark	A general-purpose distributed computing engine used for processing and analyzing a large amount of data
Struts	It helps in developing Web-based applications.
GWT	It allows Web developers to develop and maintain complex JavaScript front-end applications in Java.
Vaadin	It includes a set of Web components, a Java Web library, and a set of tools and application starters. It also allows the implementation of HTML5 web user interfaces using the Java.
PrimeFaces	A library for JavaServer Faces featuring over 100 components.
Apache Wicket	A library for creating reusable components and offers an object-oriented methodology to Web development while requiring only Java and HTML.
JavaServer Faces	A Java view library running on the server machine which allows you to write template text in client-side languages (like HTML, CSS, JavaScript, etc.).

shows the results for microservices libraries. The column *#Projects* presents the number of projects initially selected. Next, the column *Filtered* shows the number of projects removed through the filtering step. Finally, the column *Remained* presents the number of projects analyzed for each library.

3 Applicability Evaluation

In this section, we describe how we evaluated the strategy in terms of its applicability focusing on the top-9 Java libraries. Section 3.1 presents the steps to identify library experts, for example, metrics and data about classes. Section 3.2 shows an overview of our data. Section 3.3 presents the top-10 experts in each library selected in this study.

3.1 Identification of Library Experts

To evaluate the strategy in terms of its applicability, we perform three steps in this study. These three steps are described as follows.

Step 1: Extract data from source code – In this step, we obtain data from the classes created by developers from a Git repository. All data, such as added or removed LOC, written imports, commits, date, email, and developers' names, are stored locally.

Table 3. Library Descriptions

Library	Description
JavaEE	The JavaEE platform is built on top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing microservices and running large-scale, multi-tiered, scalable, reliable, and secure network applications.
Spring Boot	Pivotal solution for implementing cloudbased microservices using the well known Spring Framework.
Netflix	Netflix OSS is a set of frameworks and libraries that Netflix wrote to implement microservices in distributed-systems.
Swagger	Swagger is used to creating documentation for each microservice.
Karaf	Apache project referenced to support microservice implementations.
Spark	A lightweight web framework that has been used to implement simple and expressive microservices.

Table 4. Projects Selected for Analysis

Library	#Projects	Filtered	Reimaned
Hibernate	31,134	26,020	5,114
Selenium	19,062	17,648	1,414
Hadoop	11,715	10,778	937
Spark	9,144	7,650	1,494
Struts	4,741	4,127	614
GWT	4,086	2,635	1,451
Vaadin	3,240	2,625	615
PrimeFaces	1,881	1,401	480
Apache Wicket	1,095	896	199
JavaServer Faces	120	120	-
Total	86,218	73,900	12,318
Microservices			
Apache Karaf	264	155	109
Apache Spark	243	120	123
JavaEE	321	190	131
Netflix	653	240	413
SpringBoot	393	246	147
Swagger	357	239	118
Total	2,231	1,190	1,041

Step 2: Search for imports – From the previous step, we search for specific "imports" related to the chosen library. The idea is to explore all files that import the name of the target library. This step is performed as follows. First, the strategy gets files with all commits, for example, commits to LOC in general, comments, and mainly the header. Second, it analyzes the header of Java files containing the name of the package, all imports necessary to class, and the classes' names. Consequently, we get the "import" through regular expression pattern import+"target library", for example, "import org.apache.spark". In this example, the target library is Spark. Figure 3 shows an example of a file with data of committers. As we can observe in Figure 3, there are three attributes in this file: (1) hash code of commit, (2) name of the developer, and (3) committed source code. At the beginning of the file, there is the name of a package and many imports. In this part, our strategy is to use a regular expression to detect if the line contains the library we investigate. If the line contains the target library, we compute the hash of commit, the number of imports to the specific library, and the total imports without relation to the target library.

Step 3: Calculate skills – In this last step, we compute the skills for each developer. We rely on three metrics to identify library experts. Each metric is calculated concerning the

```
d9af8f  Developer-1 package com.huoDeveloper-.baseCount;
75b70c  Developer-2 import org.apache.hadoop.io.IntWritable;
75b70c  Developer-2 import org.apache.hadoop.io.LongWritable;
75b70c  Developer-2 import org.apache.hadoop.io.Text;
75b70c  Developer-2 import org.apache.hadoop.mapred.*;
75b70c  Developer-2 import java.io.IOException;
75b70c  Developer-2 import java.util.Iterator;
75b70c  Developer-2
75b70c  Developer-2 public class MRPersonNumCount {
58dc97  Developer-2     public static class PersonCountMap extends MapReduceBase implements Mapper<LongWritable, Text, Text, Text> {
75b70c  Developer-2
75b70c  Developer-2         private Text keyText = new Text();
58dc97  Developer-2         private Text result = new Text();
75b70c  Developer-2
58dc97  Developer-2         public void map(LongWritable key, Text value, OutputCollector<Text, Text> output, Reporter reporter) throws IOException {
75b70c  Developer-2             String lineData = value.toString();
75b70c  Developer-2
75b70c  Developer-2             String[] keyValue = lineData.split("\\t");
75b70c  Developer-2             if (keyValue.length > 1) {
75b70c  Developer-2                 String[] keyDetail = keyValue[0].split("\\|");
75b70c  Developer-2                 String[] valueDetail = keyValue[1].split("\\|");
af6d1b  Developer-3                 keyText.set(valueDetail[0] + "|" + keyDetail[1]);
af6d1b  Developer-3
af6d1b  Developer-3                 String[] callInfo = valueDetail[2].split(",");
af6d1b  Developer-3                 String[] allInfo = valueDetail[3].split(",");
af6d1b  Developer-3                 String outputCallInfo;
af6d1b  Developer-3                 try {
af6d1b  Developer-3                     long callDistance = Long.parseLong(callInfo[0]);
af6d1b  Developer-3                     outputCallInfo = "1,call";
af6d1b  Developer-3                 } catch (java.lang.NumberFormatException e) {
af6d1b  Developer-3                     outputCallInfo = "0,call";
af6d1b  Developer-3                 }
```

Figure 3. File Example with Commits of Three Developers

number of commits to a specific library. That is, when a commit using a library is identified, the metrics were calculated. In the following, we explain the 3 proposed metrics.

Number of Commits. This metric calculates the activity of each developer through the number of commits using a particular library. Through this metric, we believe it is possible to measure the library's amount in a project that a specific developer works.

Number of Imports. This metric presents the extension of knowledge in the library. For this metric, we count all imports to the library written by a developer. Repeated imports are included. If a developer wrote two equals imports, we would consider 2 imports to the target library. Figure 3 shows an example of repeated imports. There are four imports to Apache Hadoop in this figure, so we compute 4 imports for this library. Besides, if a developer made 3 imports to the same library, we compute 3 imports, for example, we are supposed to developer made 3 imports.

```
1 import org.apache.hadoop.io.LongWritable;
2 import org.apache.hadoop.io.LongWritable;
3 import org.apache.hadoop.io.LongWritable;
```

Lines of Code. To compute this metric, we developed a heuristic to count the amount of LOC related to a specific library. First, we obtain the ratio of changed LOC by the number of all imports in the file. Then, we multiply the ratio by the number of imports related to the library. Our heuristic considers 3 attributes, the number of library imports, the number of imports in general, and the number of LOC altered by a commit related to the library. The heuristic is then computed as follows:

$$LOC = \frac{\#\text{ of LOC Altered by Commit}}{\#\text{ of All Imports}} X \#\text{ of Library Imports}$$

From Figure 3, it is possible to compute an example for this metric. A developer made a commit with hash code 75b70c and an import to "import org.apache.hadoop.io.IntWritable;" (line 2). Therefore, we compute this metric as presented above and consider 10.67 LOC related to the Hadoop library.

3.2 Overview of Dataset

From the dataset projects, we computed all commits with the libraries evaluated in this study and identified 1.5 million different developers who made commits. Figure 4 shows the number of developers for the top-9 popular Java libraries. The library, with more developers that made commits, was Selenium. This library has 811,884 developers. In contrast, Apache Wicket was the library with fewer developers: 5,440. It is important to say that these developers made at least one commit for the respective library. However, we cannot consider them all experts since a single library use may not indicate high expertise.

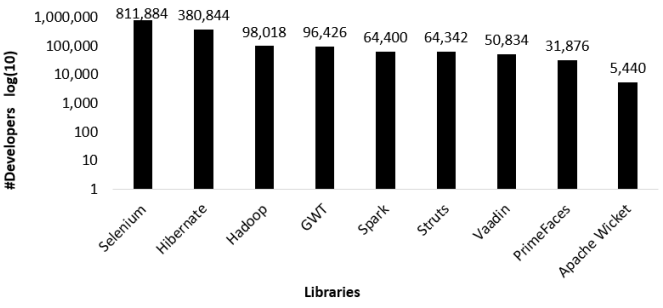

Figure 4. Number of Developers by Library

Figures 5, 6, and 7 show an overview of the metrics computed to our data set of popular Java libraries. Figure 5 presents the results to the Number of Commits per library. Figure 6 presents an overview of the metric Number of Imports per library. Finally, Figure 7 shows the results of the metric Lines of Code per library. In general, LOC (Figure 7) was the metric that presented more variation in our data set. For instance, GWT has developers that wrote more than 130 KLOC. Similarly, for Hibernate, it is possible to see an outlier developer who wrote more than 500 KLOC. In con-

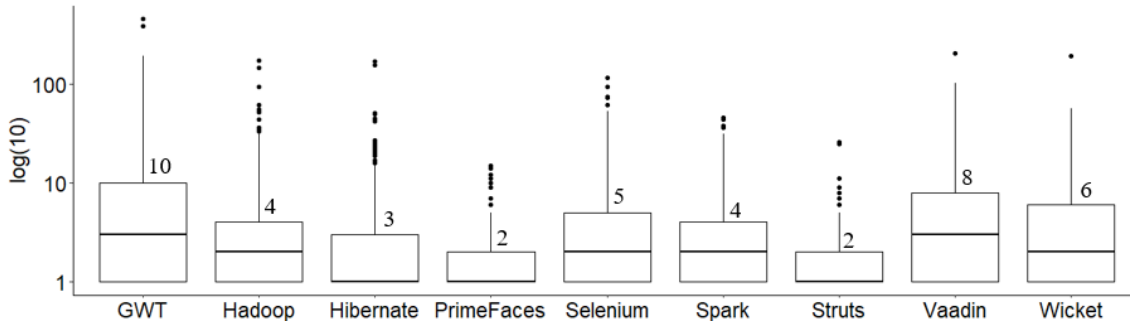

Figure 5. Number of Commits per Library

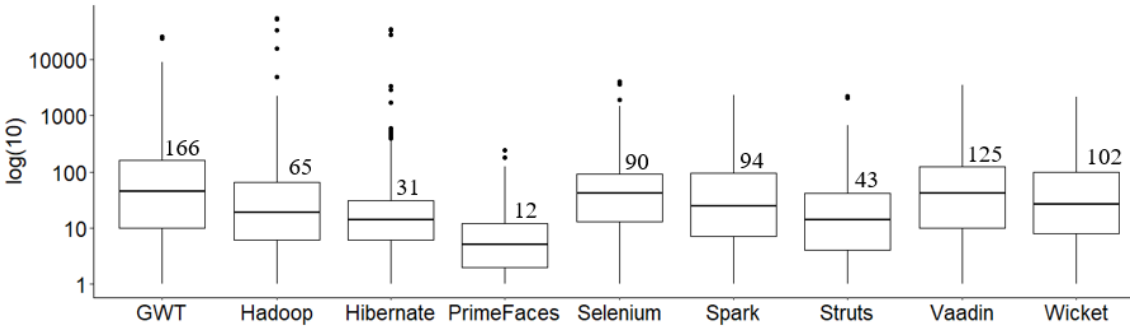

Figure 6. Number of Imports per Library

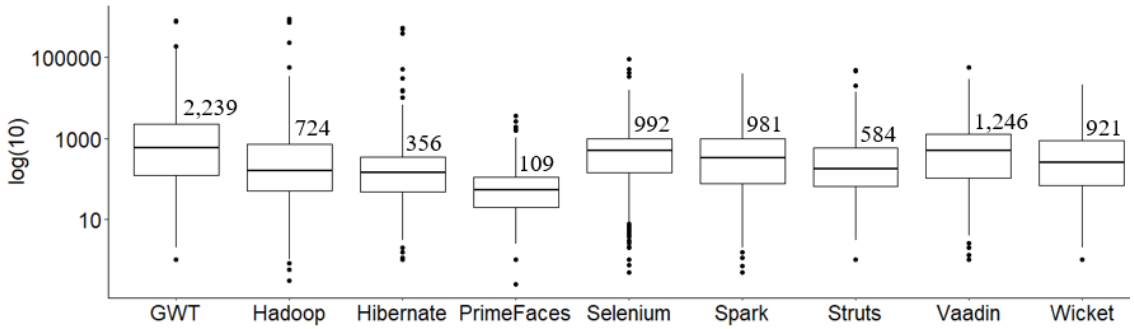

Figure 7. Number of LOC per Library

trast, some developers wrote less than 10 lines of code, for example, to the library PrimeFaces.

3.3 Top Library Experts Selection

In this section, we present the Applicability Evaluation results to verify the feasibility of library expert identification focusing on the top-9 popular Java libraries. We analyzed 16,703 software systems mined from GitHub and 9 libraries: Hibernate, Selenium, Hadoop, Spark, Struts, GWT, Vaadin, Primefaces, and Apache Wicket. Besides, we analyzed data from more than 1.5 million developers who have contributed to these projects in our dataset.

Table 5 presents the results of top library experts. To obtain these results, we aim to select the top-10 developers, but, in some cases, it was not possible to select top-10 developers. For instance, we obtained 3-top developers in library Spark. Besides, we consider a developer with a library expert only if this developer obtains high values in at least two metrics, for example, LOC & # of commit or # of imports & LOC. These developers are identified concerning their contribution. For this, we calculate the 90% percentile in each metric, then filtering the developers with any metric below this threshold (90%). This type of classification is common in other studies (Joblin et al., 2017; Ferreira et al., 2019). Finally, we sort developers by LOC (# of Library LOC). The filtering threshold was applied to remove potential false positives (i.e., developers with high # of Library LOC, but low # of Commits). In some cases, it resulted in less than 10 experts for some libraries, such as PrimeFaces (8), Spark (3), Struts (6), and Wicket (5).

In Table 5, each developer is identified by the start name of the library, followed by a sequence number (e.g., HAD (1) means the first developer expert of Hadoop). The column *# of Library Imports* refers to the metric of Number of Imports written by the developer. It counts the number of imports related to the specific library evaluated in this study. The column *# of All Imports* shows the number of imports wrote by the developer in general. When a developer wrote an import to a specific library evaluated in this study, they also wrote imports to other libraries that have not been evaluated. Hence, this metric counts all imports in relevant commits made by the developer.

The column *# of Commits* shows the results for the Number of Commits metric. This metric indicates the number of commits made by a specific developer. The column *# of LOC*

Table 5. Top Library Experts

ID	# of Library Imports	# of All Imports	# of Commits	# of LOC Altered by Commit	# of Library LOC
GWT(1)	1,693	6,836	49	637,724	157,938
GWT(2)	5,108	5,951	386	87,303	74,935
GWT(3)	4,019	5,451	452	75,700	55,813
GWT(4)	1,677	1,880	31	56,535	50,430
GWT(5)	2,497	3,714	74	54,865	36,886
GWT(6)	1,564	6,226	66	135,574	34,056
GWT(7)	2,657	6,167	71	71,767	30,920
GWT(8)	1,732	1,956	141	33,272	29,461
GWT(9)	2,249	2,558	105	31,124	27,364
GWT(10)	1,432	3,791	56	71,264	26,919
HAD(1)	15,739	32,391	172	488,882	237,550
HAD(2)	2,083	3,378	14	46,215	28,497
HAD(3)	1,024	27,277	31	476,220	17,877
HAD(4)	1,303	2,628	146	31,440	15,588
HAD(5)	932	1,518	93	16,086	9,876
HAD(6)	625	1,329	52	16,788	7,895
HAD(7)	569	1,843	55	19,899	6,143
HAD(8)	242	599	18	13,051	5,272
HAD(9)	493	617	18	6,110	4,882
HAD(10)	322	973	12	11,842	3,918
HIB(1)	3,401	5,211	155	78,781	51,417
HIB(2)	1,719	2,923	169	25,963	15,268
HIB(3)	180	432	25	24,552	10,230
HIB(4)	552	1,182	15	13,612	6,356
HIB(5)	552	791	44	7,939	5,540
HIB(6)	535	756	51	5,684	4,022
HIB(7)	509	1,281	10	9,250	3,675
HIB(8)	458	898	50	7,060	3,600
HIB(9)	202	395	17	6,880	3,518
HIB(10)	233	387	15	4,617	2,779
PRI(1)	239	16,194	6	245,319	3,620
PRI(2)	177	1,286	6	11,232	1,545
PRI(3)	72	282	15	3,500	893
PRI(4)	37	144	12	2,014	517
PRI(5)	38	545	10	6,374	444
PRI(6)	28	168	6	2,538	423
PRI(7)	28	142	6	1,904	375
PRI(8)	27	102	10	1,374	363
SEL(1)	614	820	61	8,757	6,557
SEL(2)	1,178	1,763	116	9,606	6,418
SEL(3)	707	3,166	27	27,808	6,209
SEL(4)	287	1,436	49	28,355	5,667
SEL(5)	780	1,141	93	7,245	4,952
SEL(6)	491	2,229	73	22,302	4,912
SEL(7)	242	486	18	9,513	4,736
SEL(8)	324	1,027	27	14,084	4,443
SEL(9)	394	1,095	16	12,096	4,352
SEL(10)	178	417	16	9,685	4,134
SPA(1)	757	2,208	36	22,903	7,852
SPA(2)	280	1,253	29	17,940	4,008
SPA(3)	446	834	38	7,344	3,927
STR(1)	670	3,286	3	64,468	13,144
STR(2)	531	2,432	2	24,448	5,337
STR(3)	616	2,771	2	23,419	5,206
STR(4)	175	793	9	14,753	3,255
STR(5)	133	1,076	9	21,477	2,654
STR(6)	278	818	6	7,357	2,500
VAA(1)	3,541	5,960	100	95,786	56,909
VAA(2)	561	761	46	21,537	15,876
VAA(3)	1,265	2,102	203	21,973	13,223
VAA(4)	684	4,208	74	59,710	9,705
VAA(5)	816	1,178	102	12,557	8,698
VAA(6)	510	656	31	8,913	6,929
VAA(7)	451	628	28	9,169	6,584
VAA(8)	740	1,432	30	11,746	6,069
VAA(9)	358	375	28	6,223	5,940
VAA(10)	334	495	59	8,695	5,866
WIC(1)	1,428	1,727	191	16,991	14,049
WIC(2)	1,017	1,212	55	14,255	11,961
WIC(3)	494	543	56	10,104	9,192
WIC(4)	403	451	49	9,549	8,532
WIC (5)	476	651	34	8,439	6,170

Altered by Commit presents the LOC changed by a developer when s/he made a commit related to the library (i.e., identified by a specific library import). Finally, the last column *# of Library LOC* shows the results for the metric LOC written by the developer related to the library based on our heuristic.

In this paper, the developers could be classified into Hard/Soft Committers and Hard/Soft Coders, depending on the metrics' numbers. We consider a Hard Committer when a developer obtains data equal or above 75%. That is, we use the 3rd quartile as a parameter. Hard Committers are developers who made several commits (*# of Commits*) related to the libraries which are subject of this study. For example, let us supposed that developer *Maike* made 10k commits that include hash to library Y and developer *Anna* made 1k commits using a hash to library Y. In this context, the developer *Maike* is a Hard Committer in relation to developer *Anna*. Similarly, Hard Coders are developers who wrote several lines of code related to the library (*# of Library LOC*). For instance, let us suppose that developer *Mary* wrote 8K LOC when made a commit to the library Y and developer *John* wrote 1K LOC to library Y when made a commit to the same library. Therefore, developer *Mary* is considered a Hard Coder in relation to developer *John*. Nevertheless, a developer could be Hard Committer and Hard Coder if s/he has a higher number of commits and LOC related to the library. On the other hand, we classify a developer as Soft using the same strategy to classify the Hard developers. However, we use the data below 25%, i.e., the 1 st quartile, as a parameter. Then, we discuss the below reasoning regarding this classification.

Hard Committers and Hard Coders. According to our metrics, the developer GWT (3) is a Hard Committer and Hard Coder (see Table 5). This developer made more than 450 commits and wrote more than 55 KLOC for this library. It could be noted that other developers are Harder Committer and Harder Coder. For instance, the developer HAD (1) made 172 commits and wrote more than 237 KLOC. These are some examples of Harder Committer and Harder Coder from the calculated metrics.

Hard Committers and Soft Coders. We present now the results to Hard Committers and Soft Coders. Developers HAD (1) and HAD (4) in Table 5 can be considered Hard Committers because they made 172 and 146 commits, respectively. The difference between HAD (1) and HAD (4) is only 26 commits. However, developer HAD (4) is considered as Soft Coder concerning developer HAD (1) because HAD (1) wrote more than 235 KLOC while the developer HAD(4) wrote about 15 KLOC. Developer HAD (4) wrote only 6% LOC of the developer HAD (1). Therefore, HAD (4) is a Hard Committer and Soft Coder.

Soft Committers and Hard Coders. Concerning the Soft Committers and Hard Coders, we can observe that developers PRI (1), PRI (2), SEL (1), and STR (1) in Table 5 are Soft Committers because they made only a few commits. Developer STR (1), for instance, made only 3 commits, but s/he wrote more than 13 KLOC. Therefore, this developer is considered a Soft Committer and Hard Coder.

Soft Committers and Soft Coders. As the name suggests, this category includes the developers that fewer commits and made fewer lines of code compared to their peers. For instance, Developers HIB (9), HIB (10), SEL (9), and SEL (10) are considered Soft Committers because they wrote less than 20 commits to libraries cited. Besides, these developers wrote less than 5 KLOC. Therefore, according to our metrics, these developers are considered Soft Committers and Soft Coders.

4 Survey with Top Libraries Experts

This section describes the survey applied to GitHub developers to evaluate the strategy with respect to the top-9 popular Java libraries. Section 4.1 presents the details regarding the survey developed. Section 4.2 presents a summary of some relevant findings. Section 4.3 presents the results to RQ1 regarding the Number of Commits metric. Section 4.4 presents the results to RQ2 about the Number of Imports metric. Section 4.5 presents the results to RQ3 regarding the LOC metric.

4.1 Survey Design

According to Easterbrook et al. (2008), survey studies are used to identify the characteristics of a population and are usually associated with the application of questionnaires. Besides, surveys are meant to collect data to describe and compare or explain knowledge (Pfleeger and Kitchenham, 2001). We selected the library experts with the best values in the evaluated metrics to validate them through a survey. We designed and applied a survey with the top developers identified by our strategy. We selected developers with the top-20% highest values in at least two (out of three) metrics.

We created a questionnaire on Google Forms[3] with two parts: the first one was composed of 5 questions about the background of the expert candidates; the second part also had 5 questions about the knowledge of the expert candidates regarding the evaluated libraries. Table 6 contains the tag <*libray name*> meaning a specific library, for instance, Hadoop. Also, this table shows the possible answers to the survey questions.

Table 6. Survey Questions on the Use of the Libraries

ID	Questions
SQ1	How do you assess your knowledge in <*libray name*>? ( ) 1 ( ) 2 ( ) 3 ( ) 4 ( ) 5
SQ2	How many projects have you worked with <*libray name*>? ( ) 1 to 5 ( ) 6 to 10 ( ) 11 to 20 ( ) More than 20 projects
SQ3	How many packages of <library name> have you used? ( ) A few ( ) A lot
SQ4	How often do your commits include <*libray name*>? ( ) A few ( ) A lot
SQ5	How much of your code is related to <*libray name*>? ( ) Few of my code is related to <*libray name*> ( ) My code is partially related to <*libray name*> ( ) Most of my code contains <*libray name*>

To obtain the email used by the developer to perform the

[3]https://www.google.com/forms/

commits in the source code, we used the Git-Blame[4] tool. The emails were collected to send the survey. We sent an email to developers asking them to assess their knowledge of each library. For instance, the developers were invited to rank their knowledge (Table 6, SQ1) using a scale from 1 (one) to 5 (five), where (1) means no knowledge about the library; and (5) means extensive knowledge about the library. Questions are not mandatory because they may require knowledge of the exceptional features of the library. Therefore, participants are not forced to provide an answer when they do not remember a specific library element, such as the time of development using the library and the approximate frequency of commits that contain the library. The survey remained open for 15 days in January 2019.

In summary, we present the precision evaluation results based on a survey with expert candidates in each of the top-9 popular Java libraries. The goal of this evaluation is to verify the precision of the library expert identification. We empirically selected 1,045 developers among the top-20% values in at least 2 metrics. The questionnaire was sent in January 2019. After 15 days, we obtained 137 responses resulting in a response rate of about 15%. We asked the 137 developers about their software development experience in general (background) and the use of the specific libraries investigated in this paper.

4.2 Overview

In this section, we present an overview of some relevant findings of the popular Java libraries.

Table 7 presents an overview of the experts' candidates contacted to answer our first survey. This table has the following structure. The first column (Library) indicates the name of the analyzed library. The second column (Emails sent) shows the number of emails collected and sent to expert candidates. The third column (Invalid email) presents the number of invalid emails returned by the server. The fourth column (Remaining emails) indicates the number of valid emails. The fifth column shows the number of answers we obtained for each library. Finally, in the last column, we show the response rate of each library.

Table 7. Top 20% from Library Experts Selected to Answer the Survey

Library	Emails sent	Invalid email	Remaining email	# Answers	%
GWT	160	18	142	31	22%
Hadoop	181	33	148	11	7%
Hibernate	155	10	145	16	11%
Spark	138	19	119	11	9%
Struts	42	2	40	9	23%
Vaadin	107	18	89	15	17%
PrimeFaces	30	1	29	9	31%
Wicket	23	2	21	8	38%
Selenium	209	31	178	27	15%
TOTAL	1,045	134	911	137	15%

Concerning the participants' background and replication package, we create a Web page with more details (Oliveira et al., 2020). It is worth mentioning that half of the respondents graduated in Computer Science, and 7% holds a Ph. D. degree. Concerning time dedicated to software development, 47% has more than 10 years of experience, and only 2% have less than 1 year of experience. Therefore, we can conclude that, in general, the participants are not novices.

Our study also shows that a significant amount of expert candidates makes commits. When writing code related to a specific library, they perform many imports of particular libraries and writes lines of code about the library. We support this affirmation through metrics that evaluate the amount of LOC written by a developer when they performed a commit. Table 8 shows the results of the knowledge that surveyed developers claim to have in each library. If we analyze the data about the precision of the strategy from the sum of levels 3, 4, and 5 of the Likert-type scale, we obtain on average 88.49% of precision about the knowledge of the developers, i.e., identification is correct in more than 88% of the cases. On the other hand, although a score of three may represent acceptable knowledge, if we followed more conservative criteria, only classifying as library experts the developers that informed a higher ($\geq$ 4) knowledge on the libraries obtain average, 63.31% of precision. This way, we conclude that less than 2/3 of the identified expert candidates identified by the strategy contain high knowledge about the evaluated libraries.

> About 63% of the library experts who answered the survey have high knowledge about the evaluated libraries.

Table 8. Level of Knowledge in Each Library

Library	Likert scale					Total	3-4-5	4-5
	1	2	3	4	5			
GWT	1	1	4	9	16	31	94%	81%
Hadoop	0	1	3	4	3	11	91%	64%
Hibernate	1	3	6	3	3	16	75%	38%
Spark	0	1	4	2	4	11	91%	55%
Struts	2	2	1	4	0	9	56%	44%
Vaadin	0	2	5	3	5	15	87%	53%
PrimeFaces	0	0	4	4	1	9	100%	56%
Wicket	1	0	2	4	1	8	88%	63%
Selenium	0	1	4	13	9	27	96%	81%

4.3 Level of Activity

In this section, we answer the first research question.
RQ1– *How to evaluate the level of activity of a developer in a library?*

To answer this research question, we asked the library experts the following question. ''How often are your commits related to the <*libray name*> library''? Figure 8 shows the results of this question in the first line in each chart to each library. For most libraries, the majority of the participants answered they made ''few'' commits using the evaluated libraries. This way, if we evaluated the results obtained for this label, it is possible to see that from 137 experts, 54% made ''few'' commits. For instance, in the library Hibernate, 87% of developers said they made few commits related to this library. Another library that deserves special attention is Struts. In this library, 88% of the developers responded that they made few commits. Regarding the label ''a lot'', only 39% of experts polled said they performed many commits. GWT was the library with a higher rate of answers to this

[4]https://git-scm.com/docs/git-blame

label (62%). Therefore, results indicate that the metric *Number of Commits* needs to be combined with other metrics to achieved conclusive results about the skill of developers and even develop other metrics to identify the level of activity ability.

Answer to RQ1. A large proportion of library experts make ‘‘few’’ commits using the library. Therefore, we concluded that the solo use of the number of commits could not identify library experts.

4.4 Knowledge Intensity

In this section, we answer the second research question.
RQ2– *How to evaluate the knowledge intensity of a developer in a library?*

Regarding the number of imports to indicate a library expert, we ask the developers the following question: ‘‘How often do you include an import of <*libray name*> library in your commits?’’. Figure 8 shows the results of this question from the second line in each chart to each library. We analyze the number of imports performed by developers. The main reason for this analysis is to evaluate the feasibility of inferring the skills of the developers from the types of written imports. In general, the label ‘‘few’’ and ‘‘a lot’’ are tied or with little difference between them. For example, Hibernate, Spark, and PrimeFaces have practically tied. These libraries did not show significant differences; the difference was only 1 absolute point in some cases. In only three cases, the label ‘‘a lot’’ remained significantly higher: GWT (83%), Vaadin (67%), and Selenium (78%).

From 137 experts, 68% said that they made ‘‘a lot of imports’’. However, the number informed by the experts indicates that this metric requires a combination with other metrics to achieve better results because 32% of experts said they made few imports to libraries evaluated. Therefore, from the survey results, the metric *Number of Imports*, as well as the metric *Number of Commits*, are not able to identify library experts when we apply one at a time.

Answer to RQ2. The metric *Number of Imports* is not able to identify library experts, when we use it alone.

4.5 Knowledge Extension

In order to evaluate the metric *Lines of Code*, we present the third research question as follows.
RQ3– *How to evaluate the knowledge extension of a developer in a library?*

In this research question, we analyze the developers skill from the number of LOC related to the library. We evaluate the number of LOC implemented by a developer to a specific library. For this purpose, we asked the library experts from the survey the following question. ‘‘How much of your code is related to the <*libray name*> library when you perform a commit?’’. Figure 8 shows the results to this question in the third line in each chart to each library. The libraries GWT, Wicket, Selenium, and Hadoop, for instance, obtained 74%, 71%, 70%, and 64% respectively to label ‘‘a lot’’.

We noted, however, the label ‘‘a few’’ also remained at a high level in some cases, for instance, the libraries Struts (88%) and Spark (55%). In fact, the library Hibernate remained tied to labels ‘‘a few’’ and ‘‘a lot’’. In general, from 137 experts, 39% said they write ‘‘a few’’ LOC and 61% write ‘‘a lot’’ LOC with respect to libraries. Therefore, it is possible to infer that the metric *Lines of Code* alone also does not provide indications about developer skills, although this metric achieved better precision than the metric *Number of Commits*.

Answer to RQ3. According to our analysis, the metric *Lines of Code* alone cannot reliably provide indications about developers’ skills. In general, our metrics are not feasible to identify library experts. However, our strategy is able to reduce the search space of library experts. Therefore, a company or project open source can be select a developer from a group selected by our strategy.

5 Survey with Microservices Experts

In order to favor the generalization of our findings, we did a second survey with developers of microservices libraries. For this, we conducted a selection of libraries in this domain.

5.1 Survey Design

We select the library experts to this survey in a similar way to the survey presented in Section 4.1. We created a questionnaire on Google Forms[5] in order to evaluate the knowledge of the developers about microservices libraries. The first question request the login of the developer at GitHub. This login is necessary to map the answer of the developer with our data. We ask developers about their knowledge in all six libraries of microservices. For this, we show all libraries investigates in this survey (6 libraries of microservices). We request the developer to rank their knowledge in these libraries in four levels: No knowledge, Low knowledge, Medium knowledge, and Extensive knowledge. Each level of knowledge has meaning. No knowledge: this library was never used in any project I am involved in. Low knowledge: I never used this library, but it has been used in projects I am involved in. Medium knowledge: I used this library in some projects before, but I do not master all its API. Extensive knowledge: I used this library many times, and I know a lot of its API. Table 9 shows the template of the survey.

Table 9. Level of Knowledge in Microservices Libraries

Library	No knowledge	Low knowledge	Medium knowledge	Extensive knowledge
Apache Karaf	◯	◯	◯	◯
Apache Spark	◯	◯	◯	◯
JavaEE	◯	◯	◯	◯
Netflix	◯	◯	◯	◯
Spring Boot	◯	◯	◯	◯
Swagger	◯	◯	◯	◯

We selected the library experts with the best values in the evaluated metrics to validate them through a survey. We designed and applied the survey with the top developers iden-

[5]https://www.google.com/forms/

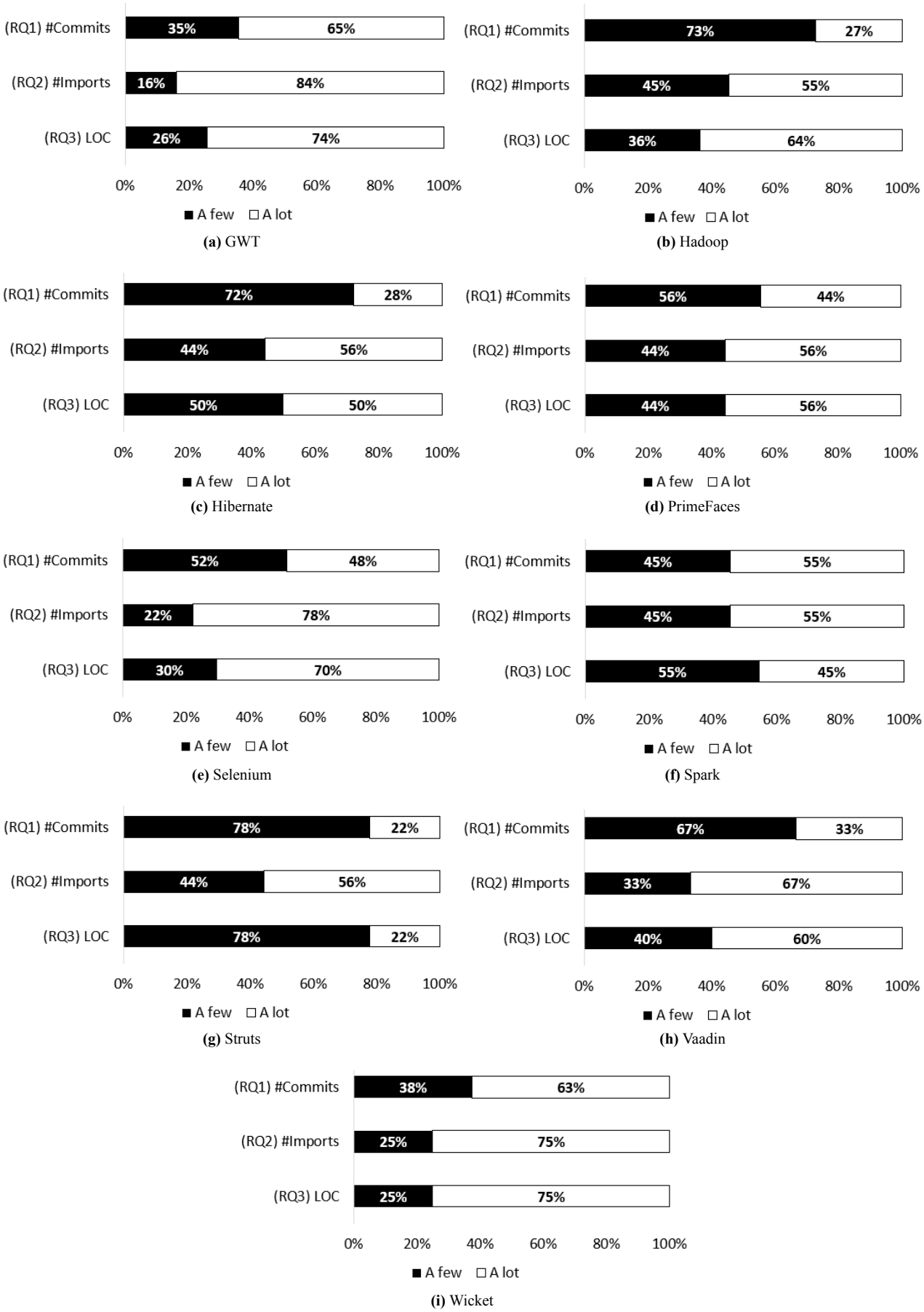

Figure 8. Results of the survey questions for each library

tified by our strategy. That is, we selected developers with the top-20% highest values in at least two (out of three) metrics. Therefore, we choose 136 candidates library experts in microservices. Figure 9 presents an overview of the number of developers by the library. The library with more candidate experts identified was Netflix with 64, and the library with fewer candidate library experts identified was Karaf with only 1.

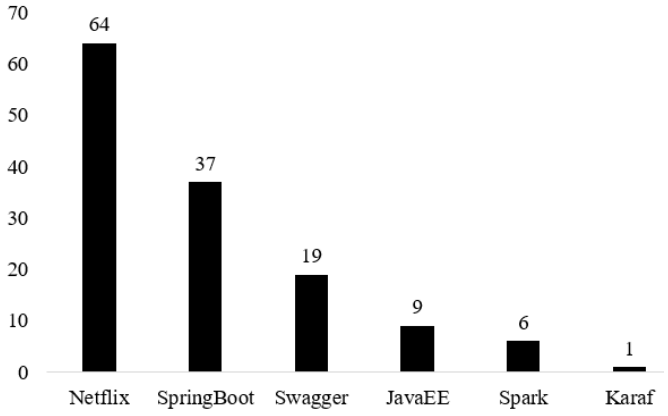

Figure 9. Number of Developers by Library in Microservices

Table 10 presents an overview of the candidate experts contacted to answer our survey. We sent 136 emails, but 38 was returned with invalid email. Therefore, we sent the survey to 98 valid emails. The library, with more amount of respondents, was Netflix with 7 candidate library experts. On the other hand, the library with fewer participants was Apache Karaf with 0.

Table 10. Top 20% from Library Experts of Microservice

Library	Emails Send	Invalid Email	Remaing Email	#Answer	%
Apache Karaf	1	0	1	0	0%
Apache Spark	6	1	5	4	80%
JavaEE	9	1	8	1	13%
Netflix	64	18	46	7	15%
SpringBoot	37	11	26	6	23%
Swagger	19	7	12	3	25%
Total	136	38	98	21	21%

5.2 Results

In this section, we present the results about a survey performed with library experts from microservices. Initially, we perform a pilot survey with 5 developers from Netflix randomly selected among the candidate experts identified. We received 3 answers for this library. From the pilot, we did not identify any problem with our survey. Then we apply the final survey for all top-20% developers with high values in at least two metrics. Note that the results of the pilot survey are part of our final results.

Table 11 shows the summary results from our second survey. The first column shows the name of the library. The second column shows the number of developers without knowledge in the library target. The third column indicates the number of developers with low knowledge in the library target. The fourth column shows the number of developers with medium knowledge in the library target. The fifth column shows the number of developers with extensive knowledge in the library target. Finally, the two last columns show the precision for medium and extensive knowledge. The library Apache Karaf is not presented in Table 11 because we did not obtain any response for this library.

Table 11. Summary Results

Library	No	Low	Medium	Extensive	Medium (precision)	Extensive (precision)
Apache Spark	2	1	1		25%	
JavaEE				1		100%
Netflix	1	1	2	3	29%	43%
SpringBoot			1	5	17%	83%
Swagger	1			2		67%
Total	4	2	4	11	19%	52%

Table 12 presents the overview of the survey applied with developers from microservices. This table has 8 columns. The column Developer represents the name of the developer. We omitted the name of developers to avoid his/her exposure. Next, six columns in the sequence represent the libraries investigates from the survey. Finally, we have the target library target name. This column represents the library that our strategy classified the developer as library experts. In this table, we have 4 scales for developers to rank her/his knowledge. The NK represents ‘‘no knowledge’’, LK represents ‘‘low knowledge’’, MK indicates ‘‘medium knowledge’’, and EK represents ‘‘extensive knowledge’’. Table 12 shows, for instance, developer-D1 was identified by our strategy as medium knowledge or extensive knowledge from library Spark. However, developer D1 answered that he/she has low knowledge in this library. D1 is an interesting case because this developer reports low knowledge in the library they had been recommended, but medium Knowledge and extensive knowledge for all others. Netflix is also an interesting case since only 3 (out of 8) reported extensive knowledge in the library, while 5 reported extensive knowledge in JavaEE and 5 in Spring Boot. On the other hand, developer-D17 was identified by our strategy as medium knowledge or extensive knowledge from library Spark, and this developer marked extensive knowledge.

From 21 developers that answered the survey, we observe that the strategy obtained a precision of 52% on average for extensive knowledge. Table 11 in the last column shows, for example, to library SpringBoot a precision of 83%. On the other hand, for library Netflix, our strategy obtained a precision of 43%. If we consider the survey results to correlate with the results of the strategy concerning only the developers that answered with extensive knowledge, we obtained 52% precision. However, if we consider developers who answered the survey with medium knowledge or extensive knowledge to correlate with strategy results, we obtain 71% precision.

6 Tool Support

We developed a prototype tool, named JExpert, to support the identification of library experts (strategy). We developed JExpert in Java programming language. JExpert currently works with Java projects, but the tool can be easily adapted

Table 12. Survey Results: Microservices (Overview)

Developer	Apache Karaf	Apache Spark	JavaEE	Netflix	Spring Boot	Swagger	Library
D1	MK	LK	MK	EK	EK	EK	
D2	NK	NK	NK	NK	LK	NK	Spark
D3	NK	MK	EK	MK	MK	MK	
D4	NK	NK	MK	NK	MK	MK	
D5	NK	NK	EK	LK	EK	MK	JavaEE
D6	LK	NK	EK	EK	EK	EK	
D7	NK	NK	EK	EK	EK	MK	
D8	MK	LK	EK	MK	EK	MK	
D9	LK	LK	LK	LK	EK	NK	Netflix
D10	LK	NK	LK	NK	LK	MK	
D11	NK	NK	LK	EK	LK	LK	
D12	NK	NK	EK	MK	MK	MK	
D13	LK	LK	EK	MK	EK	EK	
D14	LK	LK	MK	LK	EK	LK	
D15	LK	LK	MK	LK	MK	EK	
D16	NK	MK	MK	MK	EK	EK	SpringBoot
D17	NK	MK	MK	EK	EK	EK	
D18	LK	LK	EK	MK	EK	EK	
D19	LK	NK	EK	LK	EK	NK	
D20	LK	LK	EK	EK	EK	EK	Swagger
D21	LK	EK	NK	LK	LK	EK	

NK No knowledge LK Low knowledge
MK Medium knowledge EK Extensive knowledge

to identify library experts in other programming languages. JExpert is a standalone tool and runs in Windows, Linux, and MAC. JExpert is available in our website (Oliveira et al., 2020). JExpert uses static analysis to avoid Abstract Syntax Tree (AST). Therefore, it reduces the response time when analyzing large systems with hundreds of source elements, such as LOC, imports, packages, and classes. Our goal is to support recruiters with a flexible, light-weighted means to identify library experts from source code.

Figure 10 presents the simplified architecture design of JExpert. In the first moment, there are two modules: Projects and Library Name. These two modules are the input of JExpert. In other words, JExpert receives as inputs two items, (i) projects in Java that contain the target libraries, i.e., systems from a local directory informed by the user, and (ii) the names (keywords) of the libraries that a developer wants to investigate. Module Activity Extractor is responsible for extracting the code elements necessary for the computation of activities made by a developer. Besides, this module removes the old projects, i.e., projects with commits made more than three years ago, projects with less than 1 KLOC, and projects without target library.

Figure 10. JExpert Architecture Overview

From the next step, the module Developer Data Analyzer computes all data about each developer. This module is responsible for separating the number of commits to libraries and changes made from source code in general, for instance, the number of lines of code written. This module also computes the number of imports made by developers and verifies if an import is related to the target library.

The Metric Collector module computes the three metrics, as mentioned in Section 3.1. Finally, the List of Experts is generated as output with the sorted list of expert candidates from our metrics. Such a list prioritizes the library experts based on a heuristic score, i.e., higher scores come first; currently, the tool returns a ".csv" file for each library.

7 Threats to Validity

We based our study on related work to support the evaluation of a strategy to identify library experts. Regarding the assessment, we conducted a careful empirical study to assess the efficiency of the strategy from software systems hosted by GitHub. The strategy evaluated can analyze source code from platforms that follow the Git architecture. However, some threats to validity may affect our research findings. The main threats and respective treatments are discussed below based on the proposed categories of Wohlin et al. (Wohlin et al., 2012).

Construct Validity. This validity is related to whether measurements in the study reflect real-world situations (Wohlin et al., 2012). Before running the strategy, we conducted careful filtering of software systems from GitHub repositories. However, some threats may affect the correct filtering of systems, such as human factors that wrongly lead to a valid system's discard to be evaluated. Considering that the exclusion criteria to system selection were applied in a manual process, we may have discarded interesting systems that we identified as non-Java, for instance.

Internal Validity. The validity is related to uncontrolled aspects that may affect the strategy results (Wohlin et al., 2012). The strategy may be affected by some threats. To treat this possible problem, we selected a sample of 5 software systems that contain the library Hadoop from our dataset, with a diversified number of LOC. Then, we manually identified the number of commits from the GitHub

repository, the number of imports, and the number of LOC codified to the specific library. We compared our manual results with the results provided by the tool and observed a loss of 5% in metrics terms computed through the automated process. We believe that this error rate does not invalidate our main conclusions. In addition, our strategy has the goal to reduce the search space to identify library experts, that is, we do not recommend a specific developer.

External Validity. This validity is related to the possibility of generalizing our results (Wohlin et al., 2012). We evaluated the strategy with a set of 16,703 software projects from GitHub. Considering that these systems may not include all existing libraries, our findings may not be generalized. Furthermore, we evaluated the strategy with an online survey with only 158 developers that implemented projects with the investigated libraries. We analyzed the data with only 15 Java libraries. However, we chose the top libraries from the survey reported by StackOverflow in 2018, with over 100,000 responses from developers around the world. We also analyzed microservices libraries. This way, we believe these libraries can represent a reasonable option to evaluate the strategy.

8 Related Work

The use of data from GitHub to understand how software developers work and collaborate has become recurrent in software engineering studies (Greene and Fischer, 2016; Singer et al., 2013; Ortu et al., 2015; Destefanis et al., 2016; Ma et al., 2009; Begel et al., 2010; Moraes et al., 2010). Some studies seek to understand the behavior of developers concerning an interaction with their peers (Ortu et al., 2015). For example, a few studies (Ortu et al., 2015, 2016) tried to understand who are the developers with peaceful behavior and those with aggressive behavior and if these developers coexist productively in software development projects (Ortu et al., 2016). Similar studies also tried to understand if there is a relationship between bug resolution time and behavior of developers (Ortu et al., 2015). Also, some studies investigated developers manners (Destefanis et al., 2016) and seek to understand the emotional behavior of software developers (Ortu et al., 2016).

Schuler and Zimmermann (2008) investigated developer expertise based on their commit activities, which manifests itself whenever developers are using functionality. They present preliminary results for the Eclipse project. They were able to create expertise profiles that included data about what APIs a developer may be an expert in through their use of those APIs. Wu et al. (2011) proposed DREX, an approach to bug assignment using k-nearest neighbor search and social network analysis. This approach performs with the following way: 1) finding textually similar bug reports, 2) extracting developers involved in their resolution, and 3) ranking the developers expertise by analyzing their participation in resolving similar bugs. An evaluation of bug reports from the Firefox OSS project shows the social network analysis of DREX outperforms a purely textual approach, with a prediction accuracy of about 15%.

In closely related work, Greene and Fischer (2016) have developed an approach to extract technical information from GitHub developers. The work of these researchers does not differentiate developers from their level of knowledge of technical skills since a recruiter has several candidates for the same job position. Besides, such work only shows the profile of the users in GitHub, and it does not extract other characteristics of their knowledge and skills. The other limitation is that they neither provide actual data about the developer's knowledge production nor present a survey to evaluate the results. Singer et al. (2013) investigated the use of profile aggregators in the evaluation of developer skills by developers and recruiters. However, these aggregators only gather skills for individual developers, and it is not clear how they support the identification of relevant developers from a large dataset.

We believe that the strategy evaluated in our study is complementary to the described related work, providing a different approach focusing on reducing the search space to identify possible experts. Our strategy is complementary with other approaches, such as CVExplorer (Greene and Fischer, 2016). For instance, by combining our results with CVExplorer (Greene and Fischer, 2016) it is possible to select skills in the language of programming and analyze the metrics shown in our paper. To the best of our effort, we did not find a similar large scale study that evaluates some strategy able to identify library experts. Hence, we cannot compare the strategy evaluated with other studies.

9 Conclusion

In this paper, we evaluated a strategy to reduce the search space to identify library experts in software systems from source code analysis. We also presented a prototype tool that implements the strategy. The strategy evaluated is composed of three metrics: Number of Commits, Number of Imports, and Lines of Code. We assessed the strategy in two dimensions: applicability and precision. First, Applicability Evaluation analyzed the feasibility of identifying library experts candidates in large datasets. Second, Precision Evaluation compared the results provided by a strategy with developers perceptions from a survey. In total, we analyzed 16k software systems mined from GitHub, 15 libraries, and a survey with 158 developers. Our findings pointed out that the strategy was able to identify library experts in different libraries from the set of input software systems with a precision of 71% on average.

There are many possible extensions for this work. For instance, we did not consider all available data in our analysis, such as the number of forks, number of projects belonging to the developer that have received stars, the number of followers, number of methods, source code quality, and contributions to the project discussions. Besides, we did not consider the number of lines of code added and removed between versions. Future work can also extend our research to evaluate the strategy of other programming languages and libraries.

References

Alshuqayran, N., Ali, N., and Evans, R. (2016). A systematic

mapping study in microservice architecture. In *9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51.

Basili, V., Caldiera, G., and Rombach, H. D. (1994). *The Goal Question Metric Approach*. Online Technical Report.

Begel, A., Khoo, Y. P., and Zimmermann, T. (2010). Codebook: discovering and exploiting relationships in software repositories. In *32nd International Conference on Software Engineering (ICSE)*, pages 125–134.

Brown, V. R. and Vaughn, E. D. (2011). The writing on the (facebook) wall: The use of social networking sites in hiring decisions. *Journal of Business and psychology*, 26(2):219.

Capiluppi, A., Serebrenik, A., and Singer, L. (2013). Assessing technical candidates on the social web. *IEEE software*, 30(1):45–51.

Constantinou, E. and Kapitsaki, G. M. (2016). Identifying developers' expertise in social coding platforms. In *42th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 63–67.

Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in github: Transparency and collaboration in an open software repository. In *12th Proc. of the Conf. on Computer Supported Cooperative Work (CSCW)*, pages 1277–1286.

Damasiotis, V., Fitsilis, P., Considine, P., and O'Kane, J. (2017). Analysis of software project complexity factors. In *Proc. of the 2017 International Conf. on Management Engineering, Software Engineering and Service Sciences*, pages 54–58.

Destefanis, G., Ortu, M., Counsell, S., Swift, S., Marchesi, M., and Tonelli, R. (2016). Software development: do good manners matter? *PeerJ Computer Science*, 2(2):1–10.

Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311.

Ferreira, M., Mombach, T., Valente, M. T., and Ferreira, K. (2019). Algorithms for estimating truck factors: A comparative study. *Software Quality Journal*, 1(27):1–37.

Garcia, V. C., Lucrédio, D., Alvaro, A., Almeida, E. S. D., de Mattos Fortes, R. P., and de Lemos Meira, S. R. (2007). Towards a maturity model for a reuse incremental adoption. In *7th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 61–74.

Greene, G. J. and Fischer, B. (2016). Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In *31st Int. Conf. on Automated Software Engineering (ASE)*, pages 804–809.

Joblin, M., Apel, S., Hunsen, C., and Mauerer, W. (2017). Classifying developers into core and peripheral: An empirical study on count and network metrics. In *39th International Conference on Software Engineering (ICSE)*, pages 164–174.

Klock, S., van der Werf, J. M. E. M., Guelen, J. P., and Jansen, S. (2017). Workload-based clustering of coherent feature sets in microservice architectures. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 11–20.

Krüger, J., Wiemann, J., Fenske, W., Saake, G., and Leich, T. (2018). Do you remember this source code? In *40th Proc. of the International Conf. on Software Engineering (ICSE)*, pages 764–775.

Ma, D., Schuler, D., Zimmermann, T., and Sillito, J. (2009). Expert recommendation with usage expertise. In *International Conference on Software Maintenance (ICSM*, pages 535–538.

Ma, W., Chen, L., Zhang, X., and Xu, Y. Z. . B. (2017). How do developers fix cross-project correlated bugs? a case study on the GitHub scientific python ecosystem. In *39th International Conference on Software Engineering (ICSE)*, pages 1–12.

Marlow, J. and Dabbish, L. (2013). Activity traces and signals in software developer recruitment and hiring. In *16th Proc. of the 2013 Conf. on Computer supported cooperative work (CSCW)*, pages 145–156.

McCuller, P. (2012). *How to recruit and hire great software engineers: building a crack development team*. Apress.

Mockus, A. and Herbsleb, J. D. (2002). Expertise browser: a quantitative approach to identifying expertise. In *24rd Proc. of the International Conf. on Software Engineering (ICSE)*, pages 503–512.

Moraes, A., Silva, E., da Trindade, C., Barbosa, Y., and Meira, S. (2010). Recommending experts using communication history. In *2nd International Workshop on Recommendation Systems for Software Engineering*, page 41–45.

Oliveira, J., Fernandes, E., Souza, M., and Figueiredo, E. (2016). A method based on naming similarity to identify reuse opportunities. In *7th Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era - Volume 1*, pages 41:305–41:312.

Oliveira, J., Pinheiro, D., and Figueiredo, E. (2020). Web site of the paper. https://johnatan-si.github.io/JSERD2020/.

Oliveira, J., Viggiato, M., and Figueiredo, E. (2019). How well do you know this library? mining experts from source code analysis. In *18th Brazilian Symposium on Software Quality (SBES)*, pages 49–58.

Ortu, M., Adams, B., Destefanis, G., Tourani, P., Marchesi, M., and Tonelli, R. (2015). Are bullies more productive?: empirical study of affectiveness vs. issue fixing time. In *12th Proc. of the Working Conf. on Mining Software Repositories (MSR)*, pages 303–313.

Ortu, M., Destefanis, G., Counsell, S., Swift, S., Tonelli, R., and Marchesi, M. (2016). Arsonists or firefighters? affectiveness in agile software development. In *18th International Conf. on Agile Software Development (XP)*, pages 144–155.

Pahl, C. (2015). Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31.

Pfleeger, S. L. and Kitchenham, B. A. (2001). Principles of survey research: Part 1: Turning lemons into lemonade. *SIGSOFT Softw. Eng. Notes*, 26(6):16–18.

Saxena, R. and Pedanekar, N. (2017). I know what you coded last summer: Mining candidate expertise from GitHub

repositories. In *17th Companion of the Conf. on Computer Supported Cooperative Work and Social Computing (CSCW)*, pages 299–302.

Schuler, D. and Zimmermann, T. (2008). Mining usage expertise from version archives. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 121—124.

Singer, L., Filho, F. F., Cleary, B., Treude, C., Storey, M.-A., and Schneider, K. (2013). Mutual assessment in the social programmer ecosystem: an empirical investigation of developer profile aggregators. In *13th Proc. of the Conf. on Computer supported cooperative work (CSCW)*, pages 103–116.

Sommerville, I. (2015). *Software Engineering*. Pearson.

Tong, J., Ying, L., Hongyan, T., and Zhonghai, W. (2016). Can we use programmer's knowledge? fixing parameter configuration errors in hadoop through analyzing q amp;a sites. In *5th IEEE Int. Congress on Big Data (BigData Congress)*, pages 478–484.

Tsui, F., Karam, O., and Bernal, B. (2016). *Essentials of software engineering*. Jones & Bartlett Learning.

Viggiato, M., Oliveira, J., Figueiredo, E., Jamshidi, P., and Kästner, C. (2019). Understanding similarities and differences in software development practices across domains. In *14th International Conference on Global Software Engineering (ICGSE)*, pages 74–84.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

Wu, W., Zhang, W., Yang, Y., and Wang, Q. (2011). Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *18th Asia-Pacific Software Engineering Conference*, pages 389–396.

Ye, C. (2017). Research on the key technology of big data service in university library. In *13th Int. Conf. on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pages 2573–2578.