DOI: 10.36558/rsc.v13i2.8156

A Tool for Determining Developer Expertise in Specific Frameworks or Libraries

Guilherme Henrique de Assis
Universidade FUMEC
ghdeassis@gmail.com

Amanda Damasceno de Souza
Universidade FUMEC
amanda.dsouza@fumec.br

ABSTRACT

Background: In software development, the usage of frameworks is widespread to facilitate code writing. However, evaluating a developer's expertise in a specific framework poses challenges. It is difficult to determine the extent of a developer's familiarity with a framework, whether they have a deep understanding or limited knowledge. This information is valuable in various scenarios, such as during the hiring process. **Aims:** This work proposes a tool called FwkAnalyzer, which aims to analyze a developer's expertise in a specific framework or library. **Method:** The tool generates an analysis of the developer's framework usage by comparing their metrics with a benchmark. FwkAnalyzer constructs this benchmark by analyzing contributions from multiple developers in GitHub repositories that utilize the framework. To demonstrate the tool's effectiveness, an experiment was conducted with a JavaScript library. **Results:** FwkAnalyzer extracted metrics from GitHub repositories to create the benchmark, providing insights into the developers' usage of the library. Additionally, a real developer's GitHub profile was used to generate an analysis of their library usage, comparing the metrics with the benchmark. **Conclusions:** The created tool was capable of implementing the proposed work, as demonstrated with a real JavaScript library. FwkAnalyzer enables the evaluation of a developer's expertise in a particular technology, providing valuable insights by comparing their usage with other developers.

CCS Concepts

•Software and its engineering → Software libraries and repositories; •Social and professional topics → Project and people management;

Keywords

Developer Expertise, Framework, Library

1. INTRODUCTION

The field of software development is rapidly expanding and finding applications in various scenarios, leading to an increased demand for software developers [14]. Despite the continuous advancements in the state of the art of software development, it remains a challenging task that requires a high level of knowledge. Developers are constantly exposed to new technologies, components, and ideas [29].

Developer skills are crucial in team dynamics and determining developer efficiency. These skills can be categorized into technical skills, such as coding competency and quality of work, and social skills, including collaboration proficiency, project management ability, and motivation [32]. Regarding technical skills, one aspect that significantly impacts software quality and productivity is the developer's technical experience and knowledge [14].

Assessing developers' expertise is essential in various situations, such as during the hiring process, bug assignment, or contributing to a software project [18].

In software development, the usage of frameworks is widespread to facilitate code writing. A software framework is a collection of shared code with generic functionality that developers can reuse in their code to expedite the development process [10]. However, despite the common use of frameworks in software development, evaluating the depth of their usage in a specific project remains challenging. For instance, it is difficult to determine whether a project utilizes almost all the features of a framework or only a small portion.

Evaluating the extent to which a project utilizes a framework can be beneficial in certain situations. For example, during the hiring process, automatically assessing the depth of framework usage in a developer's projects can help identify their expertise in that particular framework.

Assessing the usage level of a framework in a project poses challenges. For instance, in a framework, certain commands may be used more frequently, and that is acceptable because they might be the primary commands. Thus, it is incorrect to conclude that a project does not utilize a framework effectively solely based on the absence of certain commands. Another challenge is that the importance of command groups can vary depending on the context, making generalization difficult. In light of these considerations, we formulate the following research question: **How can we assess a developer's expertise in a specific framework or library?**

To address this question, the main objective of this work is **to propose a tool capable of assessing a developer's expertise in a specific framework or library**.

FwkAnalyzer utilizes code analysis techniques and metrics generation to provide a data-driven approach for evaluating developer expertise. By analyzing developers' code contributions and tracking their usage of framework-specific commands, FwkAnalyzer generates comprehensive metrics that capture the frequency and coverage of framework usage. These metrics enable a more objective assessment of a developer's proficiency in the framework, helping organizations make informed decisions.

2. BACKGROUND

In this section, we delve into the background of the research topic, focusing on key aspects that are instrumental in assessing developer expertise in specific frameworks. We explore the concepts of Developer Expertise, Frameworks, Version Control Systems, and Software Metrics, shedding light on their interconnectedness and their role in validating the hypothesis.

The first and second subsections are discussed due to their direct relation to the research topic. As the work aims to create a tool to assess developer expertise in specific frameworks or libraries, we begin this section by introducing these concepts.

The subsequent section focuses on Version Control Systems, as they serve as the main source of data for the study. Additionally, we introduce the concept of Software Metrics, as one of the key features of the tool is its ability to generate software metrics to aid in assessing developer expertise. Lastly, in the final subsection, we discuss related work.

2.1 Developer Expertise

The term *expert* is defined by Merriam-Webster [16] as someone with special skills or knowledge representing mastery of a particular subject, derived from training or experience. In certain fields, such as individual sports, expertise can be measured more objectively using performance metrics, while in other domains, this measurement becomes more challenging [11].

Expertise is not solely innate talent but the result of dedicated application to a chosen field [5]. Moreover, achieving expertise in certain fields requires a minimum period of practice and engagement to attain higher performance [11].

Experts possess a greater quantity of domain-relevant knowledge compared to novices. However, the importance lies not only in the quantity but also in how their knowledge is organized, accessible, functional, and efficient [2]. Experts tend to organize their knowledge based on meaning, establishing stronger and more numerous links among concepts, while novices often rely on surface-level information [2].

It is crucial to differentiate expertise from experience, as significant experience does not necessarily equate to expertise. Similarly, individuals with similar expertise levels may possess different levels of experience, and vice versa [13].

Developer expertise can be assessed qualitatively, such as evaluating a developer's communication skills, as well as quantitatively, such as measuring the quality of their source code, to indicate their familiarity with specific skills and proficiency levels. These assessments find applications in various fields, including effective task allocation [36].

Software development expertise can be measured considering different aspects. For instance, Baltes and Diehl [1] defined several concepts that contribute to determining a developer's expertise: experience, knowledge, source code quality, skills, and work context.

Experience encompasses both quantity and quality. Quantity refers to the duration of experience, such as the number of years spent as a developer. Quality, on the other hand, pertains to the depth of knowledge gained during that experience, such as whether the developer has primarily worked on small projects or larger enterprise ones.

Knowledge relates to a developer's specific understanding of a technology, encompassing both depth (i.e., the level of expertise) and breadth (i.e., the range of topics covered), including algorithms, data structures, and programming paradigms.

Source code quality is a vital aspect, and according to Baltes and Diehl [1], certain properties indicate expertise, such as well-structured and readable code, as well as good performance and maintainability.

Skills, as emphasized by Baltes and Diehl [1], include communication skills, which enable developers to effectively share knowledge within a team and seek assistance when needed.

Finally, work context refers to a developer's ability to handle various situations, such as managing interpersonal relationships, working under time constraints, and dealing with ambiguous or ill-defined requirements.

The field of Developer Expertise is inherently multidisciplinary, as argued by Baltes and Diehl [1]. Addressing the challenges encountered in Software Engineering requires the study and integration of different disciplines and areas of knowledge, including Computer Science, Administration, Education, Psychology, Sociology, Linguistics, and Production Engineering. Software, being complex, modifiable, and abstract, continuously pushes the boundaries of human capacity [25].

Davenport et al. [8] explain the differences between data, information, and knowledge. Data refers to a set of distinct and objective facts pertaining to events. In an organizational context, data is often described as structured records of transactions. Information, on the other hand, is a message, typically in the form of a document or audible/visible communication, with a sender and receiver. The purpose of information is to change the recipient's perspective, and it is the recipient who determines whether a received message qualifies as information or not. Data becomes information when its creator adds meaning to it.

Assessing a developer's expertise can be seen as transforming data into information. Many studies analyze raw data produced by developers, such as source code, and generate information from it by classifying the developer's expertise level accordingly.

2.2 Framework

Despite the continuous advancements in the field of software development, it remains a challenging task that demands a high level of knowledge. Developers are constantly exposed to new technologies, components, and ideas [29]. Frameworks are utilized in these scenarios as they promise increased productivity and reduced time-to-market through design and code reuse [28].

A software framework is a collection of shared code with generic functionality that developers can reuse in their own code to expedite the development process [10]. On the other hand, a software library consists of pre-written code that developers can use to incorporate additional functionality without having to write it themselves [33].

In the software development field, the terms *framework* and *library* are often used interchangeably. However, there is a key distinction: a library typically focuses on implementing specific functionalities, whereas a framework is often composed of libraries that serve a more general purpose.

Edwin [10] describes four key features of frameworks: default behavior, inversion of control, extensibility, and non-modifiable framework code. Default behavior implies that the framework operates in a predetermined manner if not customized by the user. Inversion of control means that the control flow is dictated by the framework rather than the application. Extensibility refers to the ability of users to extend the framework's functionality by replacing default code with their own. Lastly, non-modifiable framework code indicates that users can extend the framework but cannot modify its core code. This feature ensures that the framework eases the development process by taking care of its own responsibilities, allowing developers to focus on their specific application requirements.

2.3 Version Control Systems

A Version Control System (VCS) is a system that tracks and manages changes made by software developers, facilitating the development of an evolving software artifact [37]. It enables developers to maintain a historical record of all the files in a project, which is commonly referred to as a repository [30].

VCS systems serve various purposes. One of the most widely used applications is software merging, which involves consolidating different changes made by multiple users to the same files into a unified version. VCS systems can track all the changes in the project history, storing information such as the author, date, and specific file changes. Another important feature is the ability to create software branches, allowing users to work concurrently on separate branches within the same project without interfering with each other's work. These branches can later be merged into the main branch as needed [37]. In this context, the term *commit* refers to the action of updating the repository with one's changes [30].

VCS systems offer several advantages, including speeding up the software development process, enabling collaboration among multiple individuals working on the same files without overwriting each other's work, and providing the ability to create multiple versions or tracks of the same repository [30]. Additionally, having a comprehensive history of each file is helpful in situations where it is necessary to trace and diagnose potential issues caused by recent changes.

There are two main types of VCS: centralized and decentralized. Centralized VCS systems have a single central repository, and users need a network connection to access it. In contrast, decentralized VCS systems allow each user to have a complete copy of the repository locally, enabling them to work with the repository even without a network connection. Network connectivity is only required when sharing changes with other users [37].

Git is an example of a decentralized VCS. GitHub, a platform that utilizes Git, allows users to host their repositories online and provides additional features to support social interaction within repositories [4]. One example of such interaction is the ability for users to *star* repositories they like, use, or support. The total number of stars displayed on a repository's profile serves as an indicator of its popularity.

2.4 Software Metrics

Software metrics are mechanisms used to assess attributes of a software process, product, or project [7]. They form an integral part of the software measurement process, which involves obtaining metrics from software source code by associating a characteristic with a value [19].

In this work, we employ software source code analysis to derive two specific metrics: frequency and coverage. To understand the concept of frequency, it is important to introduce the metric known as Lines of Code (LOC).

Lines of Code (LOC) or Source Lines of Code (SLOC) quantifies the number of source code lines in a software artifact [23]. KLOC is a variation of this metric, where K represents *kilo*, indicating that the scale is in thousands [24].

Frequency is employed in various contexts, depending on the concept being studied. For example, Rahmani and Khazanchi [26] utilize defect density as a measure calculated by dividing the total number of defects by the size of the software. In this work, we employ the term *frequency* with similar connotations. It is used to measure the usage frequency of framework commands. We assess individual commands as well as groups of commands, calculating their frequency by dividing the number of times they are used by the project's KLOC size.

Another significant metric employed in this work is coverage. The concept of coverage can be applied in different scenarios. In our study, we employ coverage to determine the number of commands utilized by a project and the proportion they represent out of the total. For instance, if a framework consists of ten commands and a project employs five of them, the coverage is 50%, calculated by dividing 5 by 10.

2.5 Related Work

This section presents a review of several studies that focus on the identification and assessment of developer expertise in various domains, including open-source software, recruiting, task allocation, and software engineering practices. Each study employs different techniques, tools, or algorithms and utilizes diverse types of input data and attributes to achieve its objectives.

The related studies that are more similar to the current study are the ones that primarily focus on identifying the level of framework usage in specific projects, as well as identifying library experts or developer expertise. These studies aim to obtain such information to assist in identifying proficient developers for specific scenarios.

[12] analyze keywords present in textual commit messages and associate them with developers, examining their frequency of occurrence. This approach categorizes developer expertise into unique terms (used exclusively by a specific developer), common terms (used by a group of developers), and frequent terms (used by multiple developers). The study captures expertise related to high-level packages, such as Spring Boot.

[21] utilize code metrics, including the number of commits, imports, and lines of code, to identify experts in a Java library. In another work, [20] present a tool based on this technique.

[17] search for library experts by examining repositories that employ a specific library, using metrics such as the number of commits. [31] analyze usage metrics of three Java libraries in developer commits, considering factors such

as the number of library imports and the number of lines of code related to the library. [15] employ Natural Language Processing techniques to identify software expertise from GitHub repositories. They analyze the terms present in code changes obtained from the repository history.

[34] search for library experts in Java projects by examining the repository history to determine the number of library features used by a developer, comparing it to the total number of features exported by the library. Finally, [35] propose an approach to specify skills using a specific language and then extract those specified skills from source code repositories.

[9] utilizes source code from platforms, like GHTorrent, and focuses on API usage as a key attribute. It compares new APIs with those used in the past from the data to identify experts in software libraries and frameworks among GitHub users. The study's objective is to facilitate the matching of developers with suitable open-source projects.

[18] leverages syntax patterns extracted from Python code on GitHub. The study aims to divide developers into novice and expert categories by creating a baseline and comparing developers' utilization of syntax patterns. It has implications for the recruiting process, providing insights into developer expertise based on their code's syntactical characteristics.

[3] focuses on identifying developer expertise through metrics derived from source code, including commit activity, files committed, and lines of code changed. The study compares the data obtained with StackOverflow answers, providing insights into the expertise of developers based on their commit activity in GitHub.

[6] employs machine learning techniques to calculate expertise over time. The study considers the frequency of changes to different artifacts (files or methods) in source code and analyzes the spread of changes and the time period when they were performed. The research aims to provide a comprehensive approach to identify developer expertise beyond just the total number of edits.

The reviewed studies have demonstrated various approaches to identify and assess developer expertise in different domains. A common trend observed among these studies is the utilization of source code identifiers as a valuable resource for detecting and understanding developer expertise. Techniques such as code quality analysis and static analysis have been employed to leverage the information present in source code to determine developer expertise.

The utilization of source code identifiers, combined with other data sources like bug reports, project information, and developer history, has enabled researchers to develop techniques for tasks such as matching developers with open-source projects, dividing developers into novice and expert categories, identifying technical roles, recommending developers for specific tasks, and triaging change requests.

Many existing studies focus on specific projects, limiting their applicability in different scenarios. Few studies measure developer expertise in specific technologies, such as JavaScript frameworks. To address these limitations, our work aims to develop a customizable tool that analyzes developer expertise in specific frameworks or libraries.

3. FWKANALYZER: A TOOL FOR DETECTING FRAMEWORKS USAGE LEVEL

To address the problem discussed in the previous sections, this work proposes a tool called FwkAnalyzer, which analyzes the framework usage of developers. Figure 1 provides a high-level overview of the integration of a framework into the tool (left lane) and the analysis of developers (right lane).

The tool requires a pre-generated framework benchmark as a reference to determine the level of framework usage. This benchmark is generated by analyzing GitHub repositories that utilize the framework. To integrate the framework into the tool, a list of commands that characterize the framework needs to be provided. During the analysis, metrics such as the count usage and frequency of these commands are extracted from the users' production history, obtained from their commits.

The benchmark needs to be constructed initially to enable the analysis of developers. However, it does not need to be built for every developer analysis. FwkAnalyzer can generate and store the necessary information to expedite the developer analysis process.

Using the generated benchmark, the tool can compare a specific developer's framework usage with the benchmark, providing insights into the developer's framework knowledge based on their production. The tool consists of two projects: one for generating the benchmark and performing the developer analysis, and the other for interacting with users through a web browser. Both projects were developed using the JavaScript programming language, with the first project utilizing Node.js and the second project utilizing React. Both projects are available under the MIT license, and the respective links are provided below:

- https://github.com/ghdeassis/fwkanalyzer-api
- https://github.com/ghdeassis/fwkanalyzer-web

In the following sections, we will delve into the details of benchmark construction and the developer analysis process.

3.1 Benchmark construction

Before analyzing a developer's expertise in a framework, FwkAnalyzer constructs a benchmark of the framework's usage to quantify its usage by other developers. This benchmark is created by analyzing other projects that utilize the same framework and extracting the framework-related contributions made by developers. Figure 2 provides a detailed overview of the benchmark construction process, with the following paragraphs explaining each step indicated in the figure.

Step 1 is initiated by the user who wishes to integrate a new framework into the tool. To begin the benchmark construction, FwkAnalyzer requires the following information:

- Framework and programming language name
- The commands list that the user wants to build the analysis
- File extensions that will be analyzed
- A file name to check if a project really uses the framework

Entering a list of framework commands provides flexibility to the evaluation, allowing for a focused analysis by excluding irrelevant commands based on the user's context.

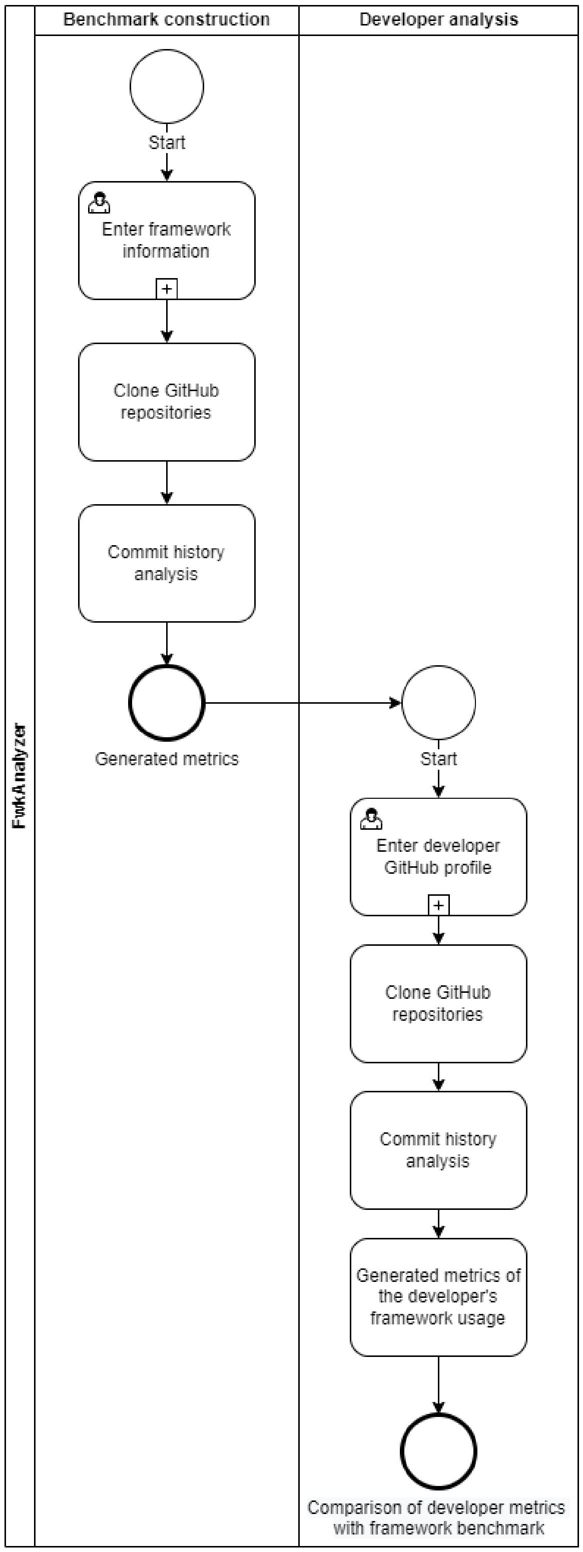

FwkAnalyzer
Benchmark construction
Developer analysis
Start
Enter framework information
Clone GitHub repositories
Commit history analysis
Generated metrics
Start
Enter developer GitHub profile
Clone GitHub repositories
Commit history analysis
Generated metrics of the developer's framework usage
Comparison of developer metrics with framework benchmark

Figure 1: FwkAnalyzer Process

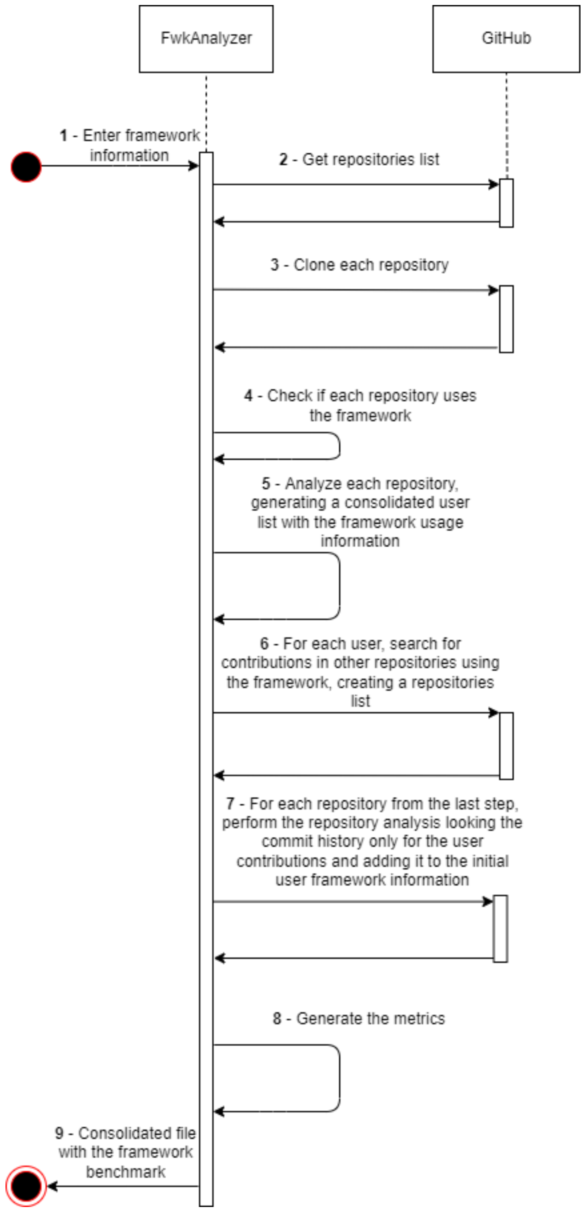

FwkAnalyzer
GitHub
1 - Enter framework information
2 - Get repositories list
3 - Clone each repository
4 - Check if each repository uses the framework
5 - Analyze each repository, generating a consolidated user list with the framework usage information
6 - For each user, search for contributions in other repositories using the framework, creating a repositories list
7 - For each repository from the last step, perform the repository analysis looking the commit history only for the user contributions and adding it to the initial user framework information
8 - Generate the metrics
9 - Consolidated file with the framework benchmark

Figure 2: Benchmark Construction Process

In step 2, FwkAnalyzer retrieves a list of 50 repositories from the GitHub public API by searching for repositories based on the framework and programming language name. Additionally, repositories are filtered based on the number of stars they have. The number of stars serves as a popularity metric on GitHub, with each user able to star a repository to indicate their interest or support. To avoid including insignificant or excessively large projects, repositories with fewer than 50 stars or more than 1000 stars are excluded. For example, to integrate the React library, which uses JavaScript as the programming language, the search query would be: *stars:50..1000 react language:JavaScript*.

According to [22], the popularity of software components, as perceived by developers, can serve as an indicator of software quality. In this regard, the number of stars on GitHub can be a helpful metric for selecting projects with a significant level of quality.

In step 3, FwkAnalyzer clones the master branch of the selected repositories and stores them in a local directory structure along with the tool. The cloning process retrieves only the Git information without downloading the repository files, enabling access to the project's change history, including the modified code and the corresponding authors. This decision optimizes the integration time for new frameworks, as the tool only needs to analyze code changes and authorship information, not every single file.

After cloning each project, step 4 verifies if the repository indeed uses the desired framework. This confirmation step is necessary because the GitHub search alone does not guarantee that a repository utilizes the framework. To perform this check, FwkAnalyzer searches for the specified file set from step 1 within the repository and checks if the framework name is present. For instance, to verify if a JavaScript project uses the React library, the tool checks if the project's configuration file, such as *package.json*, contains the name *react*.

Once the successful verifications in step 4 are completed, FwkAnalyzer proceeds to step 5, which involves analyzing the repository's history. Initially, FwkAnalyzer traverses the Git commit history, focusing solely on line additions in files with the defined extensions, disregarding deletions and untouched lines of code. In these selected lines, the tool searches for framework commands. When a command is found, the tool increments the usage count of that command for the specific developer, as what matters is the command usage by the developer rather than by the entire repository. Additionally, the number of changed lines of code is stored for each developer, specifically in files with the desired extensions. This information is accumulated as it will be important for generating metrics.

After completing this initial analysis, the tool obtains usage metrics for the framework commands specific to each developer. However, the analysis is not yet complete, as the user may have made contributions to other repositories that utilize the same framework. To identify these potential contributions, step 6 involves searching the user's commits using their email with the following query string: *author-email:example@email.com*. This search is limited to 1000 items due to GitHub constraints.

By the end of step 6, FwkAnalyzer has a list of repositories to which the user has made contributions and that utilize the framework. The verification process in step 4 is repeated for these repositories. In step 7, the tool iterates over this list and performs a similar analysis as in step 5, focusing solely on the user's contributions. The findings from this step are added to the initial user information obtained earlier.

At this stage, the initial number of 50 repositories has increased due to including the user's contributions in other repositories that use the same framework. Section 4 provides details on the final number of analyzed developers and repositories for the React library.

Upon completing the above process, step 8 involves FwkAnalyzer generating metrics using the obtained information. For example, the tool has the usage quantity of each command and the number of changed lines of code for each developer. With this information, two metrics are calculated: usage frequency and usage coverage for the framework commands.

- **Usage frequency:** the usage frequency of a command is calculated by dividing the number of times the command appears in the code by the number of lines of code (LOC). The result is then multiplied by 1000 to normalize the metric to 1000 lines of code (KLOC). This metric indicates the frequency of command usage by the developer. The equation below illustrates this calculation:

$$UsageFrequency = \frac{CommandsOccurrence}{LOC * 1000}$$

For example, if the usage frequency of a command is 15, it means that the command is used 15 times per 1000 lines of code.

- **Usage coverage:** the usage coverage of the commands is calculated by dividing the number of commands used by the developer by the total number of commands available. This metric represents the percentage of commands utilized by the developer. The equation below illustrates this calculation:

$$UsageCoverage = \frac{CommandsOccurrence}{TotalCommands}$$

For instance, if a framework has 10 commands and a developer uses 4 of them, the coverage is 0.4, which is equivalent to 40% coverage.

Usage frequency and usage coverage are calculated for each developer. At the end of the analysis, FwkAnalyzer generates the mean general coverage, mean general frequency, and frequency for each command.

- **Mean general coverage:** the mean general coverage is calculated by dividing the sum of the coverage for each developer by the total number of developers. The equation below illustrates this calculation:

$$MeanGeneralCoverage = \frac{DevelopersCoverageSum}{TotalNumberDevelopers}$$

- **Mean general frequency:** the mean general frequency is calculated by dividing the sum of all command usages by the total LOC and then multiplying by 1000. The equation below illustrates this calculation:

$$MeanGeneralFrequency = \frac{CommandsUsageSum}{TotalLOC * 1000}$$

- **Command mean frequency:** the mean frequency for each command is calculated in the same manner, but considering only the number of times that command is used. The equation below illustrates this calculation:

$$CommandMeanFrequency = \frac{CommandUsageSum}{TotalLOC * 1000}$$

These metrics will be useful during the developer analysis, as they enable a comparison between a specific developer and the benchmark. This comparison allows us to determine whether the developer's framework usage is above or below the benchmark.

The generated information is saved in a JSON file and will be referenced during the analysis of specific developers, where a comparison will be made with the previously constructed benchmark.

It is important to note that the GitHub public API has rate-limiting policies that must be followed to prevent being blocked. Users have the option to set up an access API key within GitHub and configure it in FwkAnalyzer to obtain permission for a higher number of requests. Failure to do so may result in slower processing speeds. The duration of the benchmark construction process depends on the initial number of repositories, as simultaneous requests to the GitHub API are limited. Therefore, the process may take some time. Section 4 provides more detailed information regarding the time spent on this process.

The following algorithm summarizes the steps described in the preceding paragraphs:

Algorithm 1 Benchmark Construction Process

```
Require: Framework Information
  $ReposList \leftarrow GetReposListFromGitHub()$
  for R of ReposList do Clone(R)
      if RepositoryUsesFramework(R) is true then
          $UsersList \leftarrow GetUsersList(R)$
      end if
  end for
  for U of UsersList do
      $UserReposList \leftarrow SearchUserContributions(U)$
      for R of UsersReposList do
          AnalyzeUserCommitHistory(R, U)
      end for
  end for
  $FrameworkBenchmark \leftarrow GenerateMetrics()$
```

3.2 Developer analysis

The analysis of a specific developer is conducted using a similar process to benchmark construction, but focusing on a single developer. Figure 3 illustrates the steps involved in analyzing a developer.

Step 1 begins in the FwkAnalyzer web interface, where the user can select the framework to be analyzed and enter the developer's email, which will be used to retrieve the necessary information. The framework list presented in the interface includes only the frameworks that have already been integrated into the platform.

In steps 2 and 3, FwkAnalyzer performs a process similar to steps 5 and 6 in benchmark construction. In step 2, a search is conducted on GitHub to identify the developer's commits, creating a list of repositories to which the user has made contributions and that utilize the framework being analyzed. The GitHub API search query takes into account the developer's email, for example: *author-email:example@email.com*. It is important to note that the analysis is limited to contributions made in public repositories, as access to private repositories is restricted.

In step 3, FwkAnalyzer analyzes the commit history of each repository in the list, focusing specifically on the user's contributions. During this analysis, the tool counts the occurrences of each command in the line additions of files with the specified extensions and records the number of lines of code that have been modified in those files.

In step 4, FwkAnalyzer generates the same metrics described in step 8 of benchmark construction. Once the analysis is complete, the tool provides the frequency of each command, the general frequency, and the command coverage for the developer.

In step 5, FwkAnalyzer presents a dashboard in the web interface, where the generated metrics are displayed alongside the benchmark metrics. This allows for a comparative analysis of the developer's code based on the obtained results. For instance, it is possible to compare the developer's framework usage with the average benchmark usage. Additionally, each command can be analyzed individually.

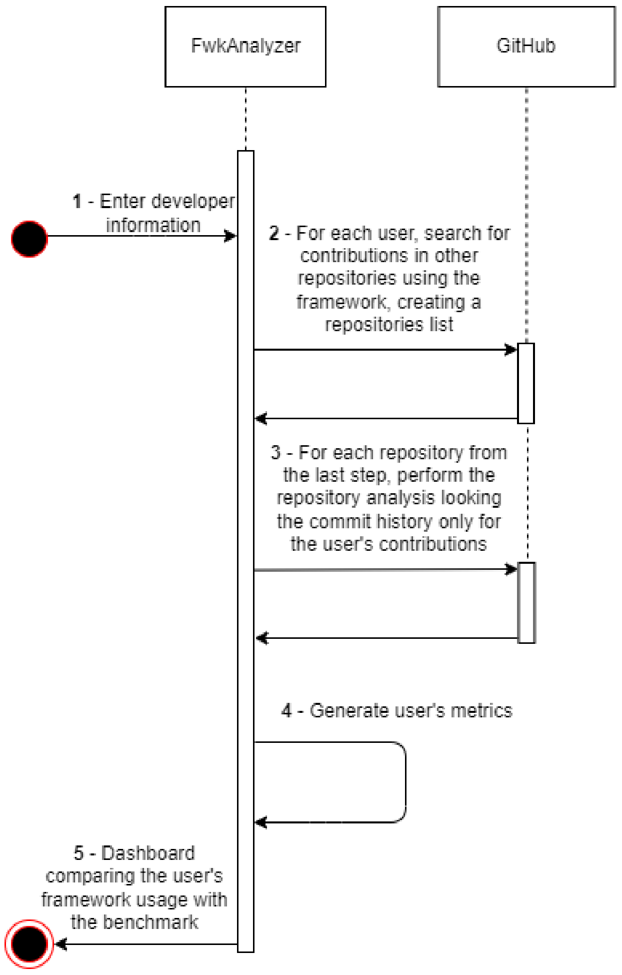

FwkAnalyzer
GitHub
1 - Enter developer information
2 - For each user, search for contributions in other repositories using the framework, creating a repositories list
3 - For each repository from the last step, perform the repository analysis looking the commit history only for the user's contributions
4 - Generate user's metrics
5 - Dashboard comparing the user's framework usage with the benchmark

Figure 3: Developer Analysis

The following algorithm summarizes the steps described in the preceding paragraphs:

Algorithm 2 Developer Analysis

```
Require: Developer Information
  $UserRepositoriesList \leftarrow SearchUserContributions()$
  for R of UsersRepositoriesList do
      AnalyzeUserCommitHistory(R, U)
  end for
  $UserMetrics \leftarrow GenerateUserMetrics()$
  CreateDashboard()
```

The generated metrics are compared to those of the benchmark to assess the developer's framework usage. Thus, at this moment, it is possible to analyze the developer framework usage. For example, if a developer's command frequency metric is higher than the corresponding metric in the benchmark, it indicates that the developer has used that command more frequently than the benchmark, proportionally to the LOC. Conversely, if the metric is lower, it means that the developer has used that command proportionally less than the benchmark.

Similarly, the general frequency metric follows the same logic but considers all the commands together. Thus, if a developer's usage of one command exceeds the mean while their usage of another command falls below the mean, the general frequency metric may still remain around the mean due to the compensatory effect.

The command coverage metric assesses the variety of commands used by the developer and is independent of frequency. For example, if a developer heavily relies on only a few commands, their coverage will be low. Conversely, if a developer uses a wide range of commands, even if it is only once each, their coverage will be high.

The results page presents these metrics in the form of charts and tables, enabling the user to draw conclusions about the developer's framework usage and gain insights into their depth of knowledge and experience with the framework. The following section discusses the insights derived from the graphs using a real example.

4. DEMONSTRATION

This section aims to demonstrate the usage of FwkAnalyzer with real information retrieved from GitHub. We have chosen the React library as an example for constructing the benchmark and conducting individual analyses. React is a JavaScript library used for building user interfaces [27].

As explained in previous sections, FwkAnalyzer requires a list of commands to generate the benchmark. The integration process allows the user to select the commands for analysis. It is not necessary to include all of the framework's commands in the list. The selection can be tailored to the user's specific context. For this demonstration, we have selected the following list of commands from the official React documentation: render, setState, Component, Fragment, componentDidMount, componentDidUpdate, componentWillUnmount, useEffect, useState, useRef and useContext.

The duration of the integration process depends on the number of repositories used for analysis. To illustrate this difference, we performed the integration process starting with ten and fifty repositories. For each developer found in the initial list, the tool searches for their contributions using the framework in other repositories, resulting in an increasing number of repositories analyzed.

Using ten repositories, the total time spent was 1 hour and 51 minutes, analyzing 46 users across a total of 388 repositories. When starting with fifty repositories, the process took 19 hours and 56 minutes, analyzing 319 users across 4238 repositories. Both cases were run on a MacBook Pro 2019 with a 2.6 GHz 6-Core Intel Core i7 processor and 16GB of RAM. The main speed limitation in this process is the interaction with GitHub, as its rate limit policy allows for 5000 requests per hour (`https://docs.github.com/en/developers/apps/building-github-apps/rate-limits-for-github-apps`) and also blocks multiple sequential concurrent requests. The initial number of repositories can be adjusted by the person running the integration process.

For this demonstration, we will use the larger dataset described in the previous paragraph. The average coverage is 51.13%, and the average frequency is 16.73 times per kLOC. Table 1 presents the average usage and frequency for each command.

For the developer analysis, we have used the author's email, which is *x@gmail.com*. Since we have set up the GitHub API Key for this email within the tool, FwkAnalyzer is able to access private repositories associated with this email. As a result, the following demonstration includes more information than what is available on the public GitHub page for this user.

To initiate the analysis, we selected React on the initial page, as shown in Figure 4. We then provided the user's email and clicked on the analyze button. The analysis process for this user was completed in 1 minute and 49 seconds.

Table 1: React Commands Metrics

Command	Average Usage	Average Frequency
render	239.93	7.81
setState	60.89	2.53
Component	318.58	7.92
Fragment	22.52	0.59
componentDidMount	19.88	0.52
componentDidUpdate	15.68	0.24
componentWillUnmount	14.65	0.28
useEffect	13.46	0.75
useState	23.09	1.12
useRef	7.10	0.32
useContext	6.52	0.14

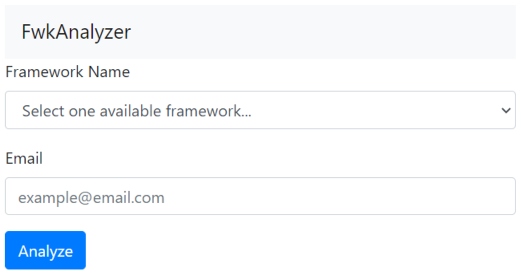


Figure 4: Initial Page

In the following paragraphs, we will analyze the dashboard generated by the tool, which is divided into figures 5, 6, 7, 8, 9, and 10. These figures showcase the insights that can be obtained from the analyzed user's data.

Figure 5 displays two graphs that provide information about command usage. The left graph shows the developer's coverage, indicating how many commands the user has used at least once out of the total number of commands analyzed. The right graph presents the general information of the framework obtained from the benchmark, showing the number of commands analyzed. By comparing the user's coverage with the general coverage, users can determine whether the specific developer has a greater coverage than the general.

Figure 6 compares the developer's command frequency with the general frequency of the framework obtained from the benchmark. The frequency is represented in terms of commands used per 1000 lines of code (kLOC). If the developer's frequency is greater than the general frequency, it indicates that the repository uses more commands per line of code compared to the general benchmark.

Figure 7 provides an overview of the command usage. It briefly shows the number of commands that have a higher

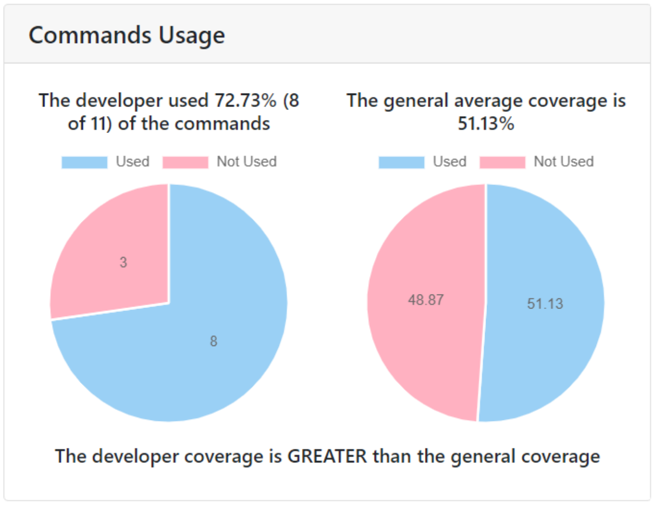


Figure 5: Commands Usage

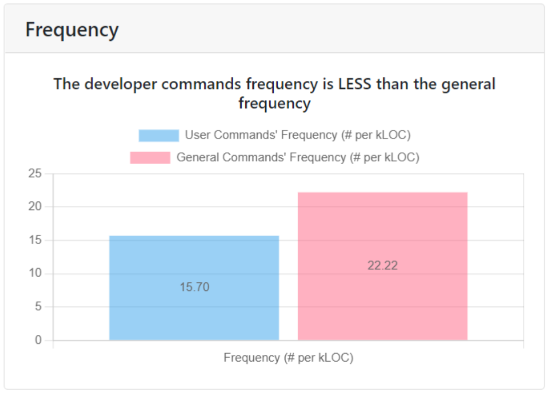
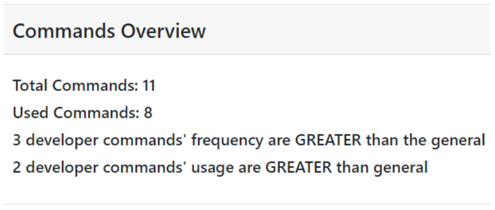


Figure 6: Frequency

frequency and usage than the general benchmark.



Commands Overview

Total Commands: 11
Used Commands: 8
3 developer commands' frequency are GREATER than the general
2 developer commands' usage are GREATER than general

Figure 7: Commands Overview

Figures 8 and 9 provide detailed graphs for each command of the framework. Each command is analyzed individually, presenting the developer's frequency and usage compared to the general framework frequency. These graphs allow for a more detailed analysis of the developer's usage of each specific command and highlight any deviations from the general benchmark metrics.

Figure 10 illustrates the developer's framework usage over the years. It displays the number of commands from the command list that the developer used in each year, providing insights into the developer's usage patterns over time.

Analyzing these graphs, we can derive several insights about the developer's framework usage compared to the benchmark. For example:

- The developer uses 72.73% of the framework commands, while the general average is 51.13%. This indicates that the developer uses more commands than the benchmark (Figure 5).
- The developer has a frequency of 15.70, meaning they use 15.70 commands per 1000 lines of code. The general mean is 22.22, indicating that the developer has a lower frequency than the general benchmark. In other words, they use fewer commands per line of code than the average (Figure 6).
- For each command, it is compared developer's commands frequency and usage with the general mean. In the command overview (Figure 7), it is observed that the developer uses 8 out of the 11 analyzed commands. Among these 8 commands, 3 have a higher frequency than the general benchmark, and 2 have higher usage than the general benchmark. Figure 8 and 9 show the generated graphs for each command, comparing its frequency and usage by the developer with the general mean.
- The graph in Figure 10 shows the developer's framework usage over the years, providing insights into the timeline of their framework knowledge, if it has been used consistently through the years, or if it was used more in a specific period. It indicates that the developer started using 183 React commands in 2022, with no records before that year.

5. CONCLUSION

The realm of programming knowledge extends far beyond being familiar with a specific framework. However, many job advertisements emphasize the requirement for expertise in particular frameworks or libraries. Assessing a developer's proficiency in a framework poses a challenge since developers typically employ only a subset of a framework's features in their work, and they may not need to utilize the entire framework.

This paper presents FwkAnalyzer, a tool designed to analyze a developer's expertise in a specific framework. To accomplish this objective, the framework must first be integrated into the tool. Integration is achieved by examining GitHub projects that employ the framework and identifying a list of relevant commands. Subsequently, metrics such as frequency and usage are generated both in a comprehensive manner and specific to each command. This data allows for a comparison between a developer's contributions to open-source projects on GitHub and the framework's metrics, thereby providing insights into the developer's framework usage.

The core idea behind FwkAnalyzer is twofold. Firstly, in order to assess a developer's knowledge of a framework, a benchmark is necessary for comparison. Without a benchmark derived from a significant number of developers who employ the same technology, it is challenging to establish an exact threshold to measure such knowledge. Secondly, FwkAnalyzer recognizes that the understanding of framework usage can vary across different contexts. Therefore,

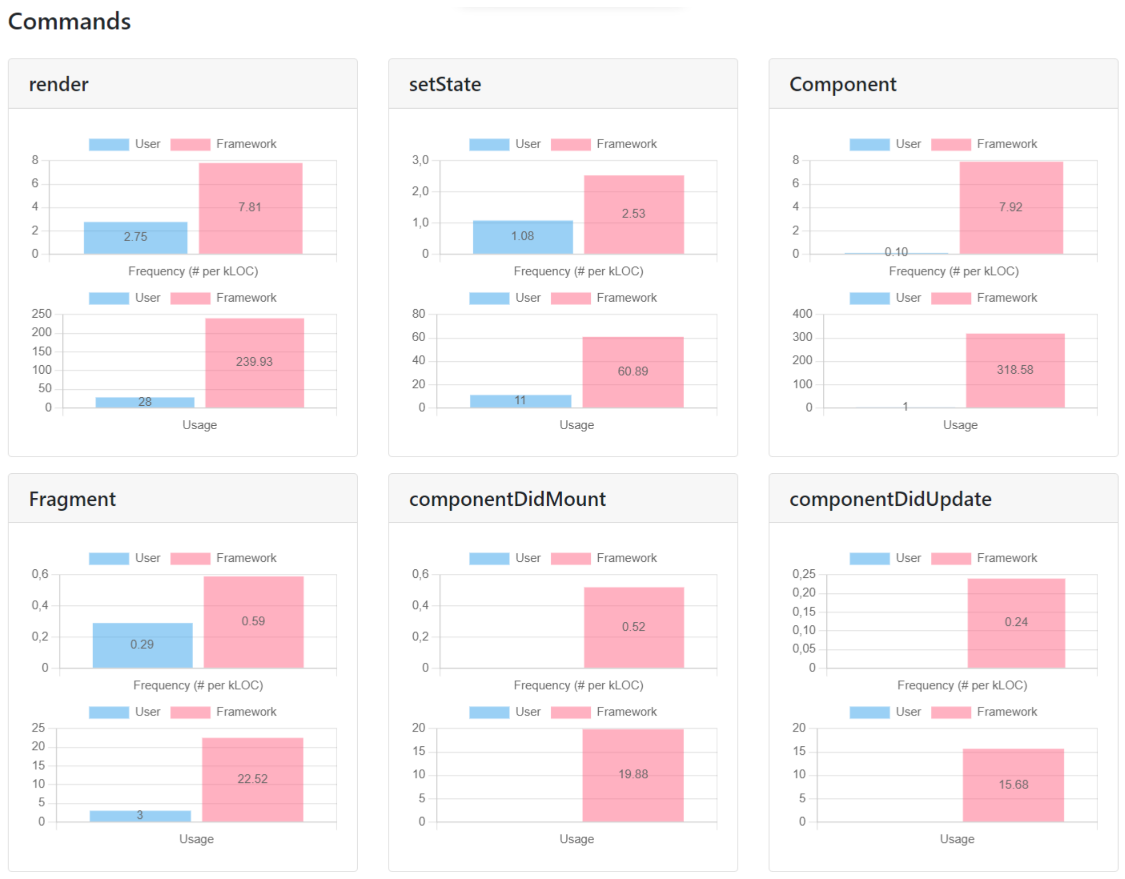


Figure 8: Commands' Analysis Part 1

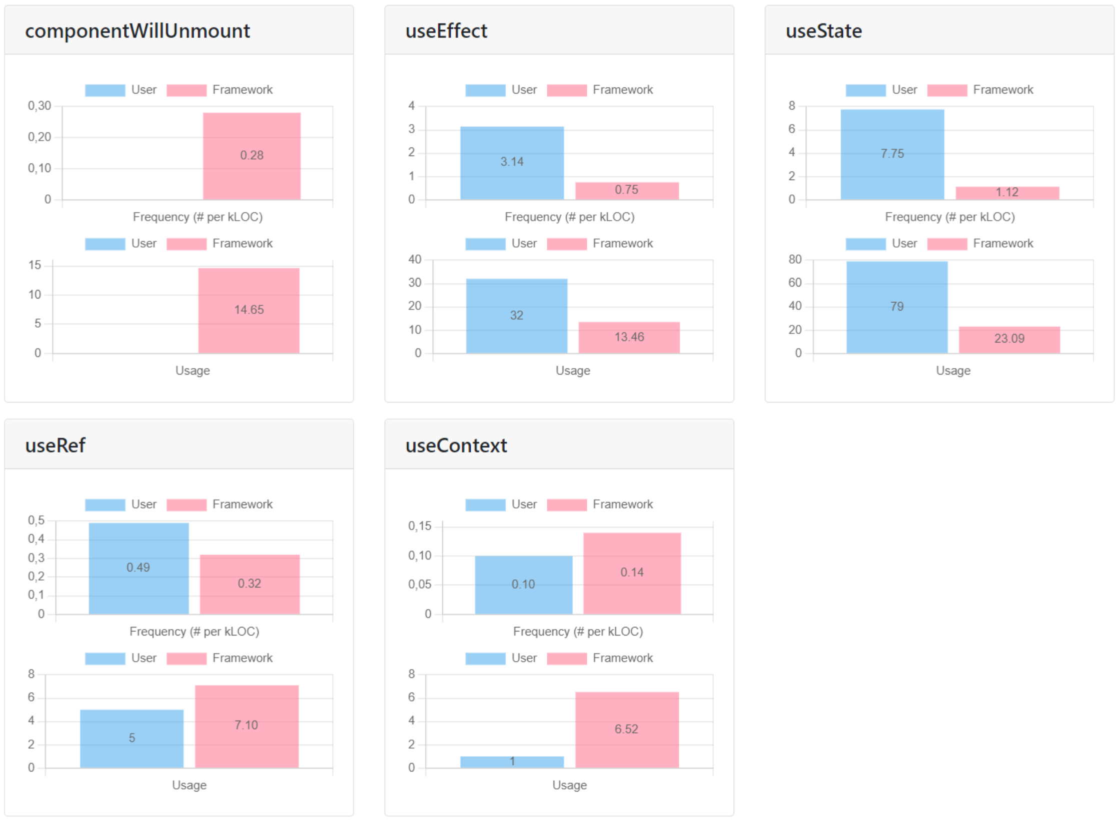


Figure 9: Commands' Analysis Part 2

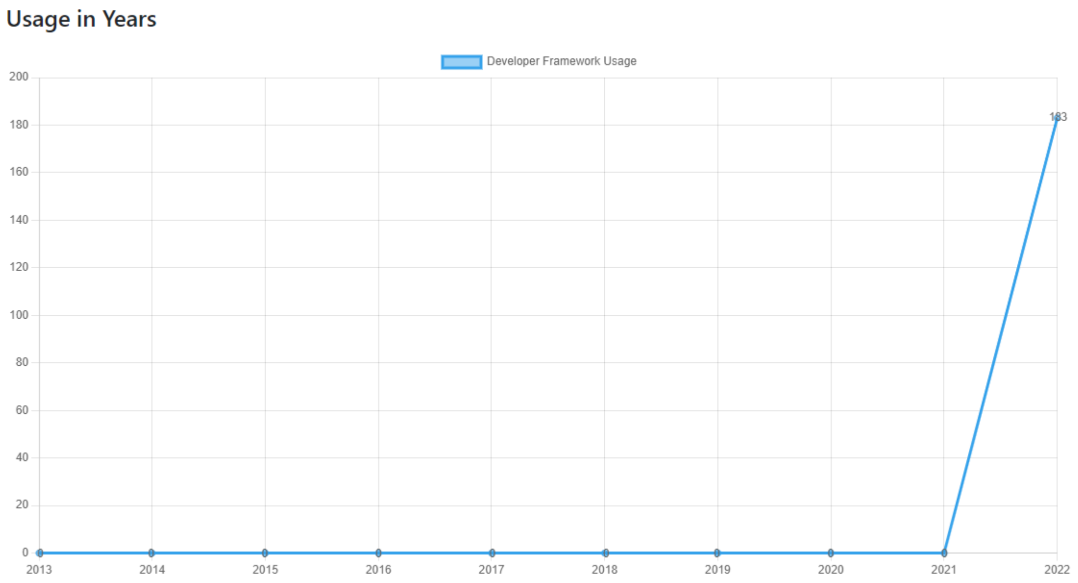


Figure 10: Usage in Years

the tool allows users to select the framework commands that are relevant within their specific context, thus constructing a benchmark tailored to their requirements.

By employing these concepts, FwkAnalyzer enables users to determine the pertinent framework commands for their specific context and construct a benchmark that reflects their needs. Once the benchmark is established, FwkAnalyzer facilitates a comparative analysis between a specific developer and the benchmark, shedding light on the developer's expertise in the framework within their particular context.

During the analysis of a specific developer, FwkAnalyzer generates graphs that provide comparisons of command usage, both in terms of individual commands and in a more general sense. These graphs offer valuable insights into the developer's expertise in the framework within the given context. Such insights can prove valuable during developer recruitment processes or when making decisions regarding project allocation.

5.1 Research Limitations

This research is subject to certain limitations in terms of data collection and metrics generation. With regard to data collection for benchmark creation, the public GitHub API imposes restrictions that limit the tool's ability to retrieve a large amount of information simultaneously. It allows for a maximum of 5000 requests per hour and blocks sequential and concurrent requests. Consequently, we had to set a reasonable number of repositories for the initial search to avoid excessively long benchmark construction times.

As a result of this limitation, the group of repositories analyzed may not provide a comprehensive representation of the entire context. Increasing the number of analyzed users could contribute to a more realistic benchmark; however, it would also extend the time required for benchmark construction. Additionally, the restriction applied to the initial search, where repositories with 50 to 1000 stars are considered, may not always yield the most appropriate group for finding developers.

Regarding metrics generation, there is a possibility of incorrect identification of framework command usage in certain situations, although it is not highly likely. This potential inaccuracy arises because the tool searches for command usage based on text occurrences, rather than performing a grammatical check. Therefore, if a command appears within a comment in the code, the tool may mistakenly count it as usage. Furthermore, the analysis is limited to the repositories' master branch and does not consider other branches that could contain valuable information.

Another limitation related to metrics generation is how FwkAnalyzer determines whether a repository uses the desired framework. The current approach does not provide a complete guarantee of framework usage within a project. Similarly, the tool does not account for specific framework versions, as it does not currently check this information and analyzes all versions together.

Finally, it is important to note that FwkAnalyzer aims to assist in assessing developer expertise in a specific framework or library, rather than providing a definitive measure of a developer's proficiency. Consequently, there are limitations that the tool cannot address. For instance, situations where a developer exclusively contributes to private repositories, which are inaccessible to the tool, pose a challenge.

5.2 Future work

Future work can address some of the limitations identified in this research. One potential improvement is to enhance the analysis process by incorporating grammar checks instead of relying solely on text matching for identifying framework commands. This could improve the accuracy of command usage identification. Additionally, the tool can be expanded to include additional metrics that provide further insights into developer expertise.

Another aspect for future work is the analysis of all project branches, rather than limiting the analysis to the master branch. This would allow for a more comprehensive understanding of framework usage within a project. Furthermore, considering framework versioning could provide valuable insights into how developers adapt to different versions and utilize specific features.

To optimize the integration process, the tool can be enhanced to support multiple configured GitHub keys, enabling concurrent requests and speeding up the benchmark construction.

Future work can involve testing Machine Learning algorithms using the data obtained from FwkAnalyzer to develop a model that can more accurately assess a developer's expertise in a specific framework.

Lastly, future work can include conducting surveys with developers and managers from real companies to gather feedback on the tool's applicability and identify areas for improvement. This feedback can provide valuable insights into the tool's effectiveness and help shape its future development.

6. REFERENCES

[1] S. Baltes and S. Diehl. Towards a theory of software development expertise. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 187–200, 2018.

[2] J. Bedard and M. T. Chi. Expertise. *Current directions in psychological science*, 1(4):135–139, 1992.

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

[4] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. A systematic mapping study of software development with github. *IEEE Access*, 5:7173–7192, 2017.

[5] N. Cross. Expertise in design: an overview. *Design studies*, 25(5):427–441, 2004.

[6] J. R. da Silva, E. Clua, L. Murta, and A. Sarma. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 409–418. IEEE, 2015.

[7] M. K. Daskalantonakis. A practical view of software measurement and implementation experiences within motorola. *IEEE Transactions on Software Engineering*, 18(11):998, 1992.

[8] T. H. Davenport, L. Prusak, et al. *Working knowledge: How organizations manage what they know.* Harvard Business Press, 1998.

[9] T. Dey, A. Karnauch, and A. Mockus. Representation

of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 995–1007. IEEE, 2021.

[10] N. M. Edwin. Software frameworks, architectural and design patterns. *Journal of Software Engineering and Applications*, 2014, 2014.

[11] K. A. Ericsson and T. J. Towne. Expertise. *WIREs Cognitive Science*, 2010.

[12] M. Hammad, H. Hijazi, M. Hammad, and A. F. Otoom. Mining expertise of developers from software repositories. *International Journal of Computer Applications in Technology*, 62(3):227–239, 2020.

[13] J. Jacoby, T. Troutman, A. Kuss, and D. Mazursky. Experience and expertise in complex decision making. *ACR North American Advances*, 1986.

[14] F. Javeed, A. Siddique, A. Munir, B. Shehzad, and M. I. Lali. Discovering software developer's coding expertise through deep learning. *IET Software*, 14(3):213–220, 2020.

[15] S. Kourtzanidis, A. Chatzigeorgiou, and A. Ampatzoglou. Reposkillminer: identifying software expertise from github repositories using natural language processing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1353–1357. IEEE, 2020.

[16] Merriam-Webster. Expert definition, 2021. Accessed: 2021-10-24.

[17] J. E. Montandon, L. L. Silva, and M. T. Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287. IEEE, 2019.

[18] A. Moradi Dakhel, M. C. Desmarais, and F. Khomh. Assessing developer expertise from the statistical distribution of programming syntax patterns. In *Evaluation and Assessment in Software Engineering*, pages 90–99. Association for Computing Machinery, 2021.

[19] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[20] J. Oliveira, D. Pinheiro, and E. Figueiredo. Jexpert: A tool for library expert identification. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 386–392, 2020.

[21] J. Oliveira, M. Viggiato, and E. Figueiredo. How well do you know this library? mining experts from source code analysis. In *Proceedings of the XVIII Brazilian Symposium on Software Quality*, pages 49–58, 2019.

[22] M. Papamichail, T. Diamantopoulos, and A. Symeonidis. User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 100–107. IEEE, 2016.

[23] B. Parareda and M. Pizka. Measuring productivity using the infamous lines of code metric. In *Proceedings of SPACE 2007 Workshop, Nagoya, Japan*. Citeseer, 2007.

[24] R. E. Park. Software size measurement: A framework for counting source statements. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 1992.

[25] R. Prikladnicki and J. L. N. Audy. Interdisciplinaridade na engenharia de software. *Scientia*, 19(2):117–127, 2008.

[26] C. Rahmani and D. Khazanchi. A study on defect density of open source software. In *2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, pages 679–683. IEEE, 2010.

[27] React. React, 2022. Accessed: 2022-08-14.

[28] D. Riehle. *Framework design: A role modeling approach*. PhD thesis, ETH Zurich, 2000.

[29] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[30] N. B. Ruparelia. The history of version control. *ACM SIGSOFT Software Engineering Notes*, 35(1):5–9, 2010.

[31] A. Santos, M. Souza, J. Oliveira, and E. Figueiredo. Mining software repositories to identify library experts. In *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 83–91, 2018.

[32] A. Sarma, X. Chen, S. Kuttal, L. Dabbish, and Z. Wang. Hiring in the global stage: Profiles of online contributions. In *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*, pages 1–10. IEEE, 2016.

[33] Techopedia. Software library, 2016. Accessed: 2022-09-27.

[34] C. Teyton, J.-R. Falleri, F. Morandat, and X. Blanc. Find your library experts. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 202–211. IEEE, 2013.

[35] C. Teyton, M. Palyart, J.-R. Falleri, F. Morandat, and X. Blanc. Automatic extraction of developer expertise. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2014.

[36] J. Yan, H. Sun, X. Wang, X. Liu, and X. Song. Profiling developer expertise across software communities with heterogeneous information network analysis. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, pages 1–9, 2018.

[37] N. N. Zolkifli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018.