university of groningen

faculty of science and engineering

# Establishing contributor roles within software repositories by mining architectural information

**Bachelor's Project Computing Science**

*July 2021*

**Author**: Kanghu Shi
**Student Number**: S3794431
**First supervisor**: Dr. Vasilios Andrikopoulos
**Second supervisor**: Anja Reuter

# Contents

## Abstract

Modern development teams are varied and diverse; they often encompass a large number of engineers with distinct specializations and technical viewpoints, working either remotely or on-site across possibly multiple locations. As such, keeping track of which developers are actively contributing/knowledgeable towards a certain piece of the architecture may pose to be a tedious, often difficult task.

In this paper, we shall propose a method (and develop an accompanying tool) which seeks to ease this process by mining architectural information from software repositories and subsequently inferring the role each contributor has within each (meaningful) part of the repository. Our resulting framework proves to be a reliable instrument in identifying experts, analyzing contribution as well as overseeing a number of valuable indicators about the development process.

# 1 Introduction

## 1.1 Background

In the distributed-systems world of today, coordination between teams is a central issue in developing software. Teams rely on one another for code, API's, documentation, updates, bug-fixes and many other aspects that require effective communication and cooperation. As with any sufficiently complex system, there are experts who often possess a deeper understanding of certain parts than others. This concept is closely related to that of code ownership, in which an expert takes primary responsibility for overseeing one or more subsystems within the codebase [1]. However, as software projects are ever-growing and increasingly becoming distributed [2], it is often not clear who these experts are and which parts of the system they have deep knowledge of [3] - [4]. When detailed knowledge of a particular piece within the system is required, the relevant expert(s) often need to be identified in a tedious, manual process, with the help of possibly multiple referrals [5]. An interview conducted with several Microsoft engineers revealed that most rely solely on connections (e.g. friends, co-workers) they already have in order to reach the people they are looking for [6], effectively reducing the search process to a game of six degrees of separation.

In this paper we propose to develop a framework for efficient identification of contributors in software repositories. By identifying, we primarily mean the process of determining each contributor's involvement within each segment of the architecture. Involvement is a nuanced term and in the context of this paper, we define it as an aggregation of several metrics measuring different qualities of contributions to the repository. A trivial, but prime example of a metric that comes to mind are the number of added lines of code (LOC). If, among hundreds of contributors, only 5% are responsible for 80% of the LOC ever committed to a particular package, that is a fairly solid sign that these people have a deeper understanding of the architectural piece in question. This particular distribution of workload isn't unheard of; the Pareto principle has been found to apply to many software engineering processes, including contribution [6]-[7]. If we need to further distinguish between the remaining 5%, the package in question may then be further split up into sub-packages and files, leading to more detailed measurements. This is, of course, a heavily simplified example of the process, yet it helps in illustrating the point. We also need to take into account time measurements (as engineers come and go), as well as other more fine-grained metrics in order to obtain truly useful insights.

## 1.2 Motivation

Our motivation stems from two main reasons, one of which we have briefly underlined above. Experts are difficult to track manually, especially when considering community-driven OSS projects (e.g. the Linux kernel) which frequently number thousands of contributors working in a semi-decentralized fashion. As identifying people who are knowledgeable on a certain issue often proves to be a time-consuming affair, developing a framework that may assist in automatizing it is certainly a topic worthy of exploration. Management often has a vested interest in determining and increasing the number of experts in order to make the project more resilient and consistent; Consequently, collective code ownership (shifting ownership towards the entire team rather than individual developers) represents a core concept in Extreme Programming [8], a type of Agile development methodology.

The second reason springs out of the observation that architectural data is ubiquitously present in online repositories. Virtually every software project makes use of Version Control Systems (VCS), which maintain a historical record of all changes made to the project since its inception. As we already have a historical log of developments and their respective authors, this rich set of data could be processed in order to infer, on a commit-basis, who the experts are over all subdivisions (i.e. files, packages) of the project.

### 1.3 Proposal

We shall propose a framework for optimizing the above-mentioned process of tracking experts of a software system and subsequently develop a tool which will serve as its implementation. Our framework roughly consists of mining all previous code-commits of an existing git repository followed by processing the acquired data into a comprehensive report which describes the contribution carried out by *each developer* over *each file* within the repository.

Contributions are measured across pre-established metrics which may be ‘collected’ from a git-commit. These metrics quantify different aspects of contributions to the codebase, such as the number of LOC, methods and comments that are added/removed or other structural changes undertaken (e.g. files deleted). Our aim is not for this report to be in a human-readable form, but rather generated under a standardized format such as JSON, essentially describing the repositories’ file structure and the contributions undertaken within each file. Visualizing the data in a meaningful way is thus delegated to the frontend, which is a part of the project in itself. We envision a lightweight frontend that should be capable of navigating the repositories’ nested structure as well as visually conveying the associated contribution metrics of each file/folder in an useful manner. Besides allowing for smooth graphical visualization of the pre-generated report, the frontend will also post-process the data to glimpse into other aspects of interest, such as team cohesion across the project.

## 2 Related work

Related work towards efficient identification of contributors within a repository may be found in areas such as quantifying contributions / source code metrics, code ownership and other alternative expert identification techniques. We shall briefly describe some of these works and how they relate with our current proposal.

Kalliamvakou et. al [7] have proposed a framework for measuring developer contribution from repository collected data. In their paper, a comprehensive list of actions that may be undertaken on a software repository is proposed and subsequently used to study the distribution of work within OSS projects. The method by which these metrics are used to infer certain aspects of the distribution resembles our approach, although we resume to a smaller set of actions due to solely focusing on code, rather than additionally considering other repository related features such as mailing list archives, bug databases and wikis. Our proposal strives towards providing an implementation that may potentially be directly applied on any given repository, thus rendering the collection of such fine-grained details as beyond the scope of this work. We do however take this option into consideration as a potential extension, which we shall delve further into during section (4.2).

Hattori and Lanza [9] put forward a classification of commits according to their associated textual comments. Their classification roughly divides commits into four categories: Forward engineering, Re-engineering, Corrective engineering, Management. Forward engineering activities are those related to incorporation of new features/requirements. Re-engineering activities are related to refactoring, redesign and other actions to enhance the quality of the code without properly adding new features. Corrective engineering solves errors, bugs or other potential disrupts in the software. Management activities are those unrelated to code in itself, such as formatting, cleaning up, and updating documentation. Their concrete method of classification consists of comparing each commit's attached comment against a predefined work bank containing words commonly associated with repository comments (e.g. 'release', 'bugfix', 'feature', etc.). Each word thus corresponds to a certain type of commit (e.g. 'bugfix' would correspond to Re-engineering). We do not adopt this approach in our project as we consider it fairly rigid, (as also pointed out by [14]), and thus inappropriate to be applied on a general basis. We do however appreciate the classification and have tried to roughly incorporate it into our implementation. Instead of inferring the type of commit from its comments, we try to assign each contributor a value for their contribution in these aspects, based on the first-hand metrics.

Teusner et. al [4] implemented a framework for expert identification using code complexity metrics. Their method aggregates multiple metrics into a single score representing the developer expertise for a particular component. The aggregation method used in their paper is Squale, proposed by Mordal et. al [11], which emphasizes improvements in badly rated system parts rather than direct metrics such as LOC. We do make use of these metrics as displaying them directly is also one of our goals, whilst utilising aggregation comes as a second source of information. Teusner et. al do indeed point out that most contributions take place on a pre-existing codebase and thus the degree of expertise could be better reflected in the quality of code produced. Although we do not currently employ any measurements about the quality of code in itself, the possibility is considered and detailed in (4.2).

## 3 Processing architectural information

In this section we shall discuss the inner workings of our framework for processing data collected from git commits. Briefly speaking, our algorithm should: given the commit history of a repository; for each distinct commit; extract all data from it that proves to be helpful in our analysis (e.g. author, nr. of modifications, files/lines affected, timestamp, etc.) From this acquired information we build an internal representation of the repository which describes both its structure as well as the computed metrics over all its internal parts. Our workflow can be roughly divided into two parts: (1) mining & processing git repositories in a consistent manner and (2) aggregating the previously collected metrics.

## 3.1 Mining repositories

For any given git repository, we may access its commit history, reflecting a log of all changes made to the repository since its inception. By traversing these chronologically, we are able to reconstruct the repositories' structure (i.e. organization into files/folders) as well as take into account any modifications undertaken by its contributors. Each modification represents a valuable source of data: it completely captures the changes made to a specific file before/after the modification. As such, we are able to collect metrics on-the-fly, by parsing each modification appropriately and storing relevant information. The metrics of folders can then be recursively inferred from their children.

Our first-hand metrics (i.e. those that are computed directly and not as a result of aggregation) are comprised of the following:

| **Metric** | **Description** |
|---|---|
| `LOC +/-` | the nr. of lines of code that have been added/removed |
| `FUNC +/-` | the nr. of methods that have been added/removed |
| `FUNC *` | the nr. of changes undertaken to pre-existing methods |
| `COM +/-` | the nr. of lines of comments that have been added/removed |
| `LOC.<f> +/-` | `LOC` that have been added/removed in files of format `<f>` |
| `Modification.MODIFY` | the nr. of files modified |
| `Modification.ADD` | the nr. of files added |
| `Modification.REMOVE` | the nr. of files removed |

Table 1: Directly collected metrics

Once these first-hand metrics have been obtained for each file, we move on towards aggregating these sources of data into composite metrics that may provide insights into other aspects of development. In order to take into account the timeframe, we maintain a separate value for each metric describing the contribution only for a recent period of time. This is required in order to allow focusing only on those developers who have recently contributed code and are well-versed with the current architecture. The timeframe may be specified as a parameter upon running the tool; otherwise a default value of 6 months will be assigned.

## 3.2 Aggregating metrics

In aggregating the different metrics detailed above, we try to roughly assign each developer a weighted score in each of the four classification types proposed in [10]. Forward engineering consists of implementing new features into the codebase and thus may be considered a function of lines of code that are added, new methods that are implemented and total number of files added. Corrective engineering and Re-engineering pertain to making changes on the existing codebase (although for different purposes). These can be described as a function of the nr. of changes applied to existing methods, as well as the number of removed methods/lines of code. Management type of activities can be captured by actions such as adding comments, or modifying documentation/configuration files (usually identified by common formats such as `.xml, .yaml`, etc.). Source code metrics such as those mentioned earlier often have different domains and scales [12], rendering simple aggregation methods such as sum-averaging inefficient towards obtaining truly useful information. In the scope of this paper, we will not delve too deeply into exploring aggregation techniques but rather explore the possibility of modelling aggregation as a linear equation. This aspect will be further discussed in the results section (5.4). Given developer $i$, his assigned score in the aspect of 'Forward Engineering' (FwD) would thus be:

$$i_{FwD} = f(i_{LOC+},\ i_{FUNC+},\ i_{MODIFICATION.ADD})$$

We may then assign weights to each developer based on how they fared in certain aspects compared to the total amount acquired by all developers. Thus, if D is the set of developers, developer $i$ will have an assigned FwD weight of:

$$w.i_{FwD} = i_{FwD}\ /\ \Sigma j_{FwD},\ \text{j in D}, j \neq i$$

If we wish to flag all developers who've fared beyond average in a particular aspect, we can compare their assigned weight to the mean weight. The following statement will be true when developer $i$ has fared better than average in FwD.

$$w.i_{FwD} > 1/(\#D)\ ?$$

It must be pointed out that we do not distinguish between Corrective Engineering / Re-engineering, but rather aggregate them under the same metric. The reason for this is due to our limited capabilities of determining whether a change on the codebase represents a fix or an improvement/optimization. However, we do consider the possibility of extending the framework with such features (more in 4.2). As such, our aggregation functions are defined in Table 2.

Where $f_1$, $f_2$, $f_3$ are simple aggregation methods generalizable as linear equations:

$$f_i(x, y, z) = w_1 \cdot x + w_2 \cdot y + w_3 \cdot z,\ \forall i \in \{1, 2, 3\}$$

Finding appropriate weights such that the result of these aggregations lend accurate information represents a goal of the paper itself. The method through which we seek to find these weights resembles linear regression - we attempt to actively "fit" the equation according to observed data. Our findings are detailed in subsection (5.4).

| **Metric** | **Aggregation method** |
|---|---|
| Forward Engineering | $f_1(i_{LOC+},\ i_{FUNC+},\ i_{MODIFICATION.ADD})$ |
| Re-engineering | $f_2(i_{LOC-},\ i_{FUNC*},\ i_{MODIFICATION.MODIFY})$ |
| Management | $f_3(i_{COM+},\ i_{LOC.<config>},\ i_{MODIFICATION.REMOVE})$ |

Table 2: Aggregated metrics

## 4 Implementation

In our implementation, we make use of `pydriller`, a Python framework for mining git repositories. This framework helps us greatly simplify the boilerplate code of the repository collection pipeline. We use `pydriller's` provided methods for traversing the commit history and then progressively feed each commit into our pipeline. Based on each modification within the commit, our dynamic structure will update itself appropriately as well as take into account the particular contribution underlying the modification. Any relevant information contained within the modification will be used towards updating the values associated with each measurement.

Metrics that are directly reflected from the nature of the commit (e.g. lines of code committed) can be gathered in a straightforward manner, whilst other values need to be computed through dedicated means. In order to determine the number of comment lines before/after each modification, regular expressions are applied on the source code. These expressions are pre-defined in the configuration file for all supported languages. We define an associated expression for each supported family of languages, as the manner through which comments are annotated is identical across languages from the same family (e.g. `C/C++/Java/C#` all use `C`-like syntax for marking comments).

```
    regex_C_like = '(?://[^\n]*|/\*(?:(?!\*/).)*\*/)'
    regex_XML_like = '<!--(.*?)-->'
    regex_PY_like = '#[^\n\r]+?(?:\*\)|[\n\r])'
```

Thus, the `C`-like regex captures all characters that come after double slashes (`//...`) or are enclosed within a slash and star sequence (`/*...*/`). Similarly, the `XML` and `PY` regular expressions capture comments as they are found in the XML family of languages (`<!--...-->`) and Python (`#...`) respectively.

The repository is internally represented as a tree, with folders as its non-leaf nodes and files as leaves. Each node will have a list of contributors representing all the developers who've ever contributed to the file/folder in question. Contributors are described by their name and contribution hash-map, which stores the values attained by the contributor within each of the measured metrics.

Our generated report will thus take the form of a single JSON object represented as a string, encapsulating the entire repositories' structure within it. The following exemplified object illustrates the core of the idea.

```
{
    "name" : "repository-name",
    "contributors" : [
        {
            "name": "contributor_1",
            "contribution": {
                "LOC+" : "100",
                // other metrics
            }
        }
    ],
    "children" : [
        {
            // nested child object
        }
    ]
}
```

## 4.1 Visualization

Visualizing the JSON object(s) described above is realized through a web frontend. The D3 Javascript library is utilized to allow for smooth creation of graphical objects & state transition. We've chosen to represent the data structure itself as a collapsible tree of nodes, with each being represented by a package/file. The metrics are displayed using bar charts, with axis dimensions bound to the names/quantities of the metrics in question. Fig. 1 illustrates the collapsible tree structure of Apache Tomcat as visualized from our application; we consider such a hierarchical structure most suitable for representing a software repository.

We must point out that our frontend does not merely visualize the data, but also performs work associated with aggregation: e.g. composite metrics & cohesion indicators. We have decided to keep these computations in the frontend due to efficiency: the generated report only contains 'raw' metrics that are directly collected; whilst aggregation is kept separate from the initial collection. Moreover, this allows us to flexibly test a wide range of values to be used in aggregation, easing the experimental process.

## 4.2 Extending the framework

In this subsection we shall discuss possible extensions to our application. As pointed out in the literature study, two potential courses of action for further extending the framework's capability are (1) *incorporating additional sources of data* (i.e. not only commits but also mailing lists, wiki activity and other information that may be gathered from software repositories) and (2) *measurements about the quality of code.* Although other alternatives exist, in

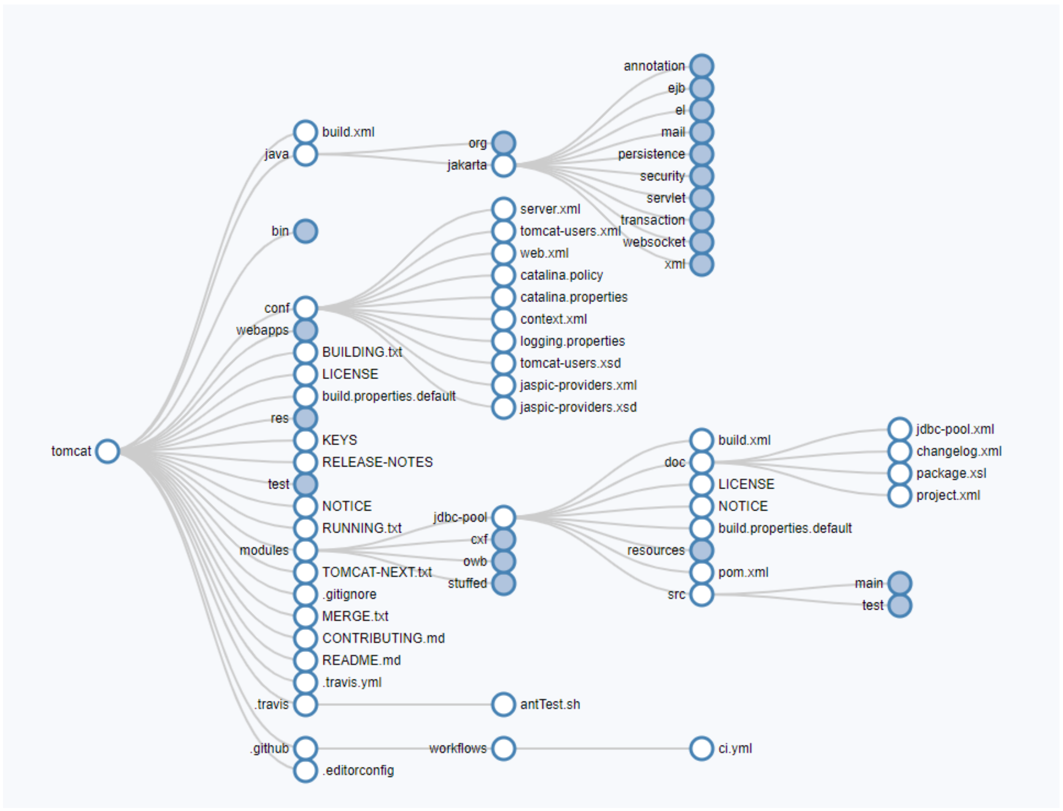

Figure 1: Apache Tomcat repository tree as visualized from our application

the scope of this project we chose to consider only these extensively due to their suitability with our framework.

1. **Additional sources of data**

   The field of Mining Software Repositories (MSR) is not confined to only source code modifications but extends to any type of information that may be gathered from software repositories. The type of information that is available depends on the repositories' organizational specifics (i.e. platforms/services used), as handling mailing archives, bugs and other repository-related data goes beyond the scope of `git` itself. Consequently, the method(s) by which these types of information may be mined also depends on the implementation used. In order to allow for as many types of repositories to be analyzed, we have chosen to not pursue this option in the project itself, but rather mention it as a possible extension.

   In the context of Github itself (which is the `git` service we are specifically interested in), additional sources of data come in the form of pull request conversations, workflow actions and documentation wikis. These can be programmatically accessed through Github's provided API. Integrating the acquired data into our framework may be done in the form of metrics which quantify the number of bugs solved, tasks accomplished (identified through Github Actions testing pipeline) and documentation created. These types of information would be highly valuable as it would permit measuring the amount of work realized by contributors beyond direct figures such as the nr. of `LOC` but rather in terms of work-items/bugfixes carried out.

2. **Quality of code**

   Another possible extension comes in the form of integrating measurements about the code itself into our framework. There are a number of aspects that provide insight into the quality of a particular code snippet/file, such as the cyclomatic complexity, method size (in LOC) and interfacing degree (e.g. number of method parameters, coupling between classes). Such values would allow us to measure the *risk profile* of commits, enabling the classification of developers according to the quality of their commited code. The method underlined here is based on the Delta Maintainability Model (DMM) detailed in this paper [15] by di Biase et. al.

## 5 Results

We have analysed repositories of several established open source projects in order to determine the degree of utility of our tool. Analysed repositories include Apache Hudi, Spring, Django and Node.JS. The use cases which we've tested our tool against are its two main intended functionalities: identification of experts (5.1) and contribution analysis (5.2). We also discuss the applicability of our cohesion indicators (5.3) and the refinement of aggregation weights (5.4).

### 5.1 Expert identification

At a first glance, the applicability of our tool seems to be promising; main developers can be quickly identified within each package. Two levels into the tree, the top 10% developers already account, on average, for more than 80% of the total committed lines of code. On the opposite spectrum of the tree, leaf nodes (files) are mostly always composed of only a few contributors accounting for over 80% of the committed code on average.

Figures (2) and (3) illustrate how the most significant contributors of a particular package or file may be identified. Whilst the figures describe the top developers from the viewpoint of the entire repository, more accurate information may be accessed if we target a specific package or file. At a first glance, "experts" of the repository can be quickly identified, although the degree of their expertise at such a broad level (the entire project) may vary. It can be seen that **Django's** distribution differs greatly from that of **Kafka**: the amount of code contributed is spread out more evenly in the latter. Thus beyond the task of identifying developers within a project, we also obtain insights into the distributiveness of tasks.

These results provide a promising glimpse into our tool's applicability. When we wish to identify contributors whose expertise lies within a particular package, the node only needs

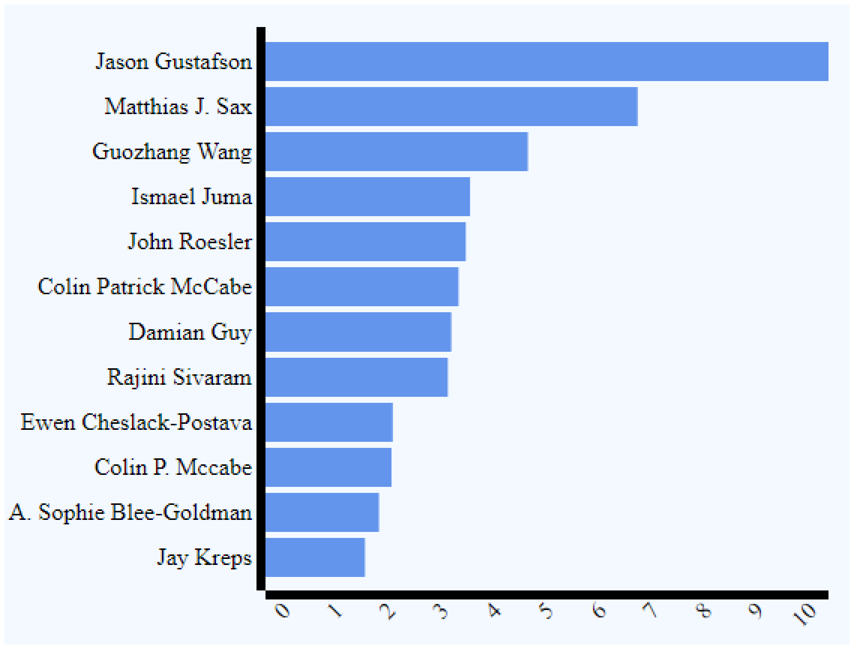

Figure 2: Apache Kafka's top 12 contributors as % of LOC committed

to be selected in order to visualize the top developers by a certain metric. If the returned list of contributors is inconclusive, we may target one of its child nodes for more precise data. The number of contributors diminishes significantly upon each level of nesting.

| Repository | avg. level-2 LOC | avg. level-2 LOC (past 6 months) | leaf LOC |
|---|---|---|---|
| Apache Hudi | 90.98% | 92.22% | 99.97% |
| Apache Kafka | 93.72% | 94.44% | 99.75% |
| Apache Tomcat | 94.55% | ≈ 100% | 99.82% |
| Django | 95.35% | ≈ 100% | 99.93% |
| Strider | 95.5% | ≈ 100% | 99.82% |
| pydriller | 84.63% | 97.78% | 99.25% |

Table 3: The percentage of LOC realized by top-10% developers of each project

The table above illustrates the average percentage of lines of code committed by the top 10% developers (as measured by `LOC+`) within different nesting levels of the repository. As level 1 is the root node itself, level 2 would thus be any package residing directly within the repository. The second column is computed in the same manner as the first, except we only take the contribution from the past 6 months in consideration (consequently, we also decide the top-10% developers solely within this timeframe). Computing a statistic such as the one exposed above may be realized from the vantage point of any other metric, yielding the specific distribution for that measurement.

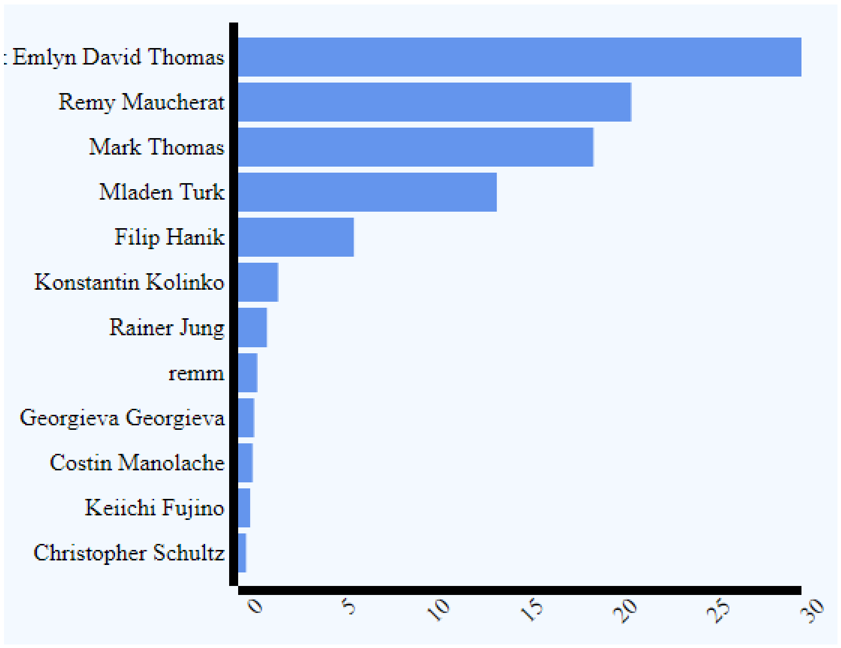

Figure 3: Apache Tomcat's top 12 contributors as % of LOC committed

## 5.2 Contribution analysis

Our tool provides a solid framework for repository contribution analysis. Although a variety of other tools exist for the purpose of analysing repositories from a contributory perspective, most of them [e.g. 18, 19, 20] are concerned with generating a chronological statistic/report of commits, but do not provide an interface for navigating the repository (thus allowing ourselves to only focus on specific packages/files). This functionality paves the way for effectively tracing the amount of effort that has been put into any node (package/file) of interest, together with the relevant list of contributors, rather than overseeing the project from a "broader" perspective.

As such, our tool may be employed on a wide range of repository-analysis use cases, such as keeping track of the contribution put into the project by any specific contributor. Fig. 4 represents the contribution of Jason Gustafson, main developer of Apache Kafka, as visualized from our frontend. The metrics are shown as a percentage of the total collective contribution, thus revealing important information about the amount of effort put in by any participant to a repository. Such a feature might prove to be an useful tool for assessment purposes by an educator or corporate management team.

## 5.3 Cohesion & Task distribution

Remarkably, our analysis goes beyond that of the contributory/identificatory perspective. By further inspecting the generated report, we are able to infer certain indicators about the overall development process, such as the average distribution of contributions attained across the project by a certain percentage of developers. Such figures lend us valuable information

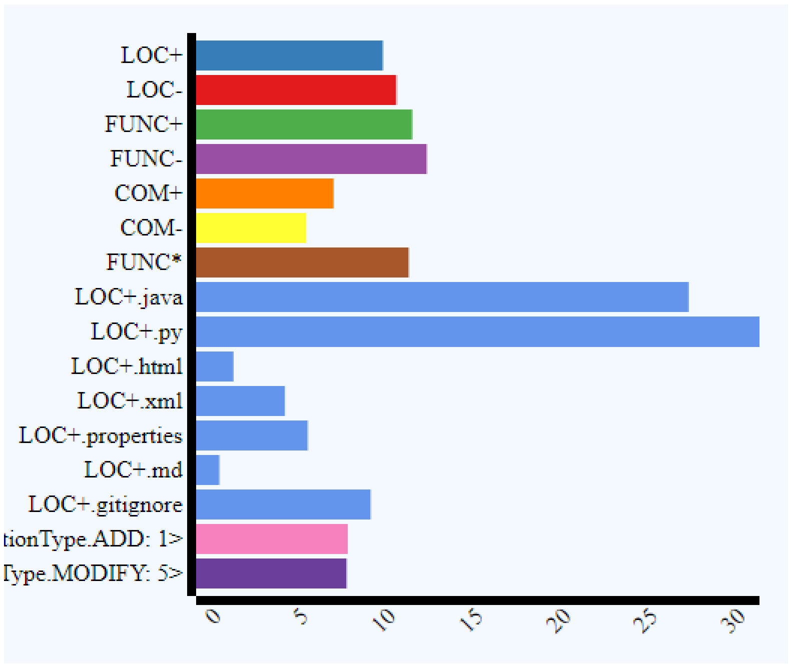

Figure 4: Individual contribution of Jason Gustafson to Apache Kafka's repository, in quality of main contributor

about aspects such as code/method ownership, essential indicators in Extreme Programming (XP) or Agile practices [21].

The table below (4) describes our findings w.r.t the cohesion & task distribution of our analysed repositories. The values above represents the amount of work realized by top 10%

| **Repository** | `LOC+` | `LOC-` | `FUNC+` | `FUNC-` |
|---|---|---|---|---|
| Apache Hudi | 97.97% | 87.74% | 48.47 % | 33.99% |
| Apache Kafka | 95.5% | 89% | 40.23% | 35.97% |
| Tomcat | 99% | 98.92% | 16.44% | 16.48% |
| Strider | 99.69% | 60.92% | 33.22% | 21.88% |
| pydriller | 97.53% | 91.36% | 18.82% | 19.18% |

Table 4: Distribution of metrics acquired by top 10% developers, as measured by LOC+

developers of each analysed repository (as measured by `LOC+`) in other metrics of interest (e.g. methods implemented). For example, in the case of Apache Tomcat, developers responsible for almost 99% of the total amount of `LOC` account for only 16.44% of the methods present within the project. We should also point out that our definition of `LOC` includes any type of "code" found within the repository, including configuration files (e.g. `.properties`, `.json`, etc.) and documentation.

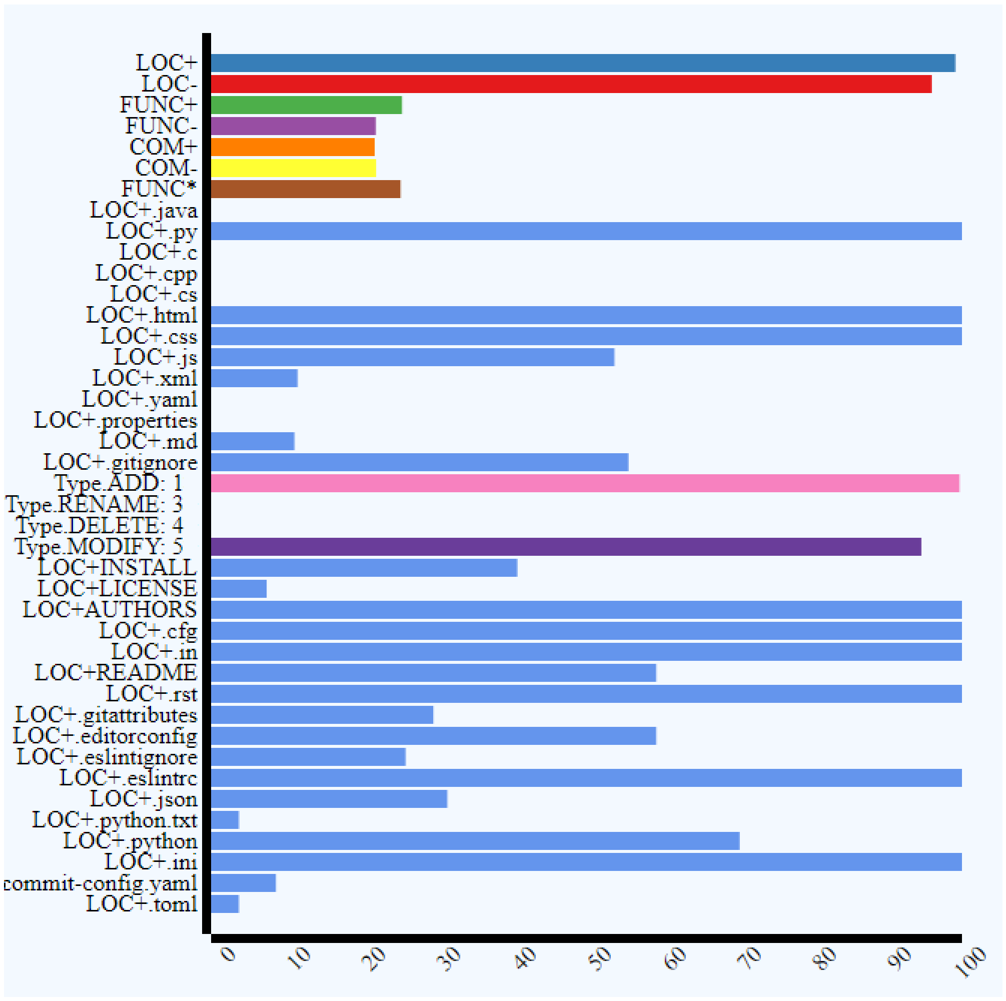

Figure 5: Django's distribution of metrics acquired by top-10% developers as measured by LOC+

## 5.4 Aggregation weights

In the process of aggregating first-hand metrics towards classifying the role of each contributor within a particular node (as per the classification described in [10]), we employ a method which vaguely resembles linear regression. However, in a classic regression scenario, we try to fit a pre-existing set of values to a number of explanatory variables; whereas in our case we do not possess values which quantify developers contribution in different areas already but rather attempt to create the value ourselves. As such, possibly the best source of "observed values" would be the publicly available lists of contributors of our analysed repositories and their known attributions. Additionally, we attempt to scale each metric according to the set of aggregated values, such that no metric heavily outweighs the others in computing the final result.

We will analyse the case of Apache Tomcat, as its list of developers and their associated main attributions are easily accessible on the project's website [19]. Specifically, we will focus on three contributors for which we roughly know the assumed roles (table 5).

| **Contributor** | **Role** |
|---|---|
| Mark Thomas (markt) | CGI, SSI, WebDAV, bug fixing |
| Costin Manolache (costin) | Catalina, Connectors |
| Filip Hanik (fhanik) | Clustering, Release Manager |

Table 5: Developers of Apache Tomcat and their attributions, as presented on the Tomcat website

We shall now present the aggregated development measurements (i.e. classification according to Engineering, Re-engineering, Management) of the contributors highlighted above. From the observed list of roles, we roughly expect Mark/Costin to fare high in the area of *Engineering* (due to development attributions), Filip/Mark in the area of Re-Engineering (bug fixing and release patches) and Filip in tasks related to Management

As can be seen from Figs. 6, 7 and 8, the resulting metrics roughly resemble our initial assumption. The method of aggregation used in computing these graphs is that of simple addition between terms (metrics) with no additional weights. Although this approach certainly yields some data about the nature of each contributor's role within the repository, the method of aggregation requires further refinement until more fine-grained information can be obtained.

One possible approach for stabilizing the (aggregation) values further is that of scaling the coefficients appropriately, such that no particular measurement heavily outweighs the others. Here we present the measured metrics for the developers highlighted above, specifically those who have been directly used in aggregating their roles.

| **Contributor** | `LOC+` | `LOC-` | `FUNC+` | `FUNC-` | `FUNC*` | `.ADD` | `.MODIFY` |
|---|---|---|---|---|---|---|---|
| markt | 297661 | 160394 | 7721 | 3097 | 29064 | 1177 | 20424 |
| costin | 12136 | 1225 | 71 | 49 | 198 | 25 | 39 |
| fhanik | 97131 | 29087 | 4866 | 680 | 8347 | 347 | 1959 |

Table 6: Metrics of Apache Tomcat developers used in aggregating their qualitative contribution. We have chosen to denote ModificationTypes by a dot (.) for brevity

As can be observed, `FUNC+`, `FUNC-`, `FUNC*`, `.ADD` and `.MODIFY` mostly yield values between 1 and 2 scales of magnitude lower than measurements about the lines of code (`LOC+/-`). Here, the "weight" of each action may be arbitrarily decided on a case basis; we have decided not to incorporate any pre-defined weight as that would exceed the scope of this project, but

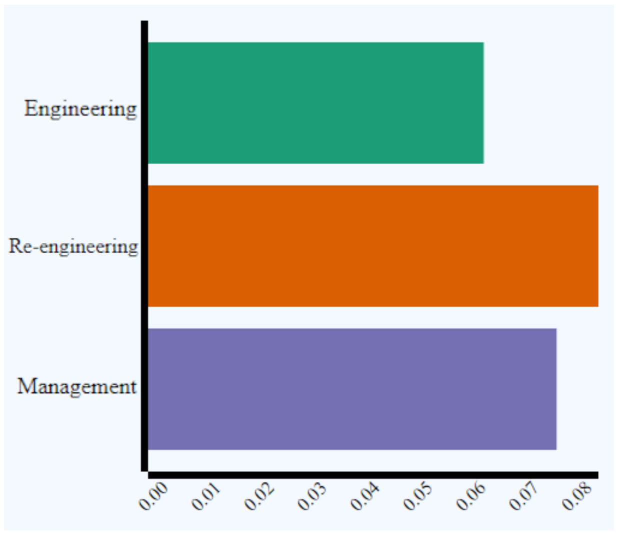

Figure 6: Filip's aggregated metrics within Apache Kafka

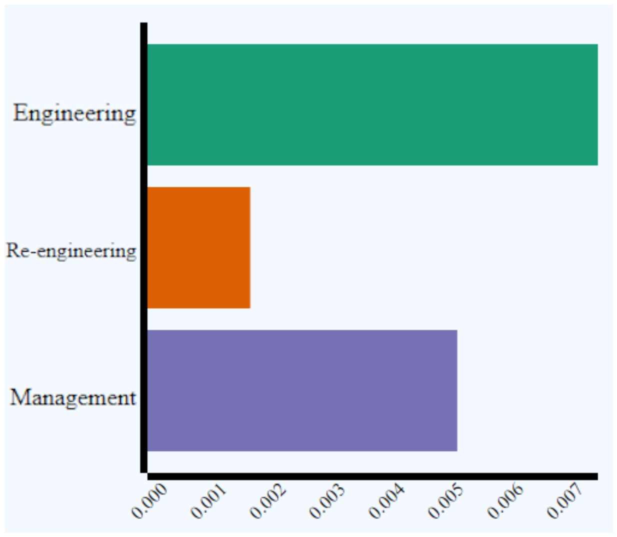

Figure 7: Costin's aggregated metrics within Apache Kafka

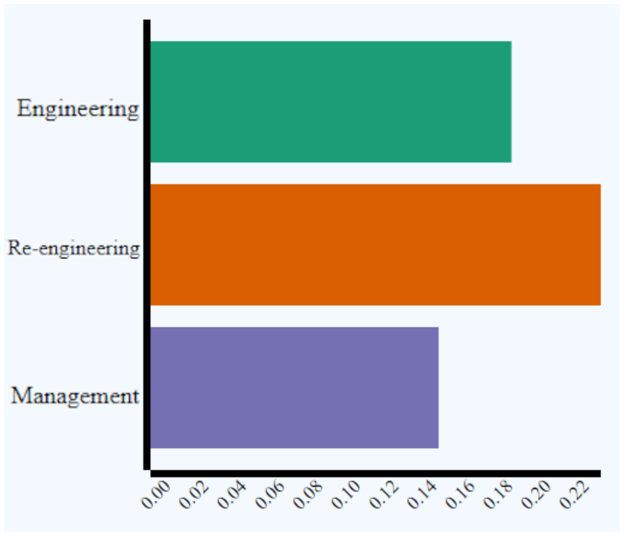

Figure 8: Mark's aggregated metrics within Apache Kafka

rather restrict ourselves to discussing the possible options. Whilst the first 4 measurements quantify lines of code, and thus may be considered already under the same "dimension" (it is a matter of deciding whether lines of code added within a method are worth "more" than lines of code added elsewhere towards aggregating a particular role), the latter 3 represent "modifications" and might be scaled up, since, in general, a modification (be it within a method or the entire repository) will affect more than a single line of code and thus should weigh more in the aggregation strategy.

## 6 Conclusion

Our tool proves to be a reliable framework for monitoring a repository from a multitude of perspectives, ranging from contribution to expertise and cohesion. The degree of analysis provided is extensive and may be applied in a wide range of use cases (detailed in subsection 5). A number of notable works in the field have proven to be influential towards our project, most importantly the classification put forward by Hattori and Lanza [9] on which we've based the aggregation technique.

Although our proposal is not in itself a novel idea (contribution analysis is a major subfield of MSR), the resulting tool provides a fresh approach for repository analysis, especially due to the navigable interface & clear visualization. We are confident that our approach has considerable potential, both in direct applicability (analysis of pre-existing repositories) and research potential. A possible line of work could be further extending our framework with features such as the ones described in subsection 4.2. Such additions could render our application into a hallmark tool for contribution analysis & expert identification. Moreover, the manner in which we have designed our application's components (i.e. independent repository analyzer & web frontend) allows for smooth integration into a new or pre-existing web application. The underlying framework for generating reports may be exposed under an API service, thus permitting any potential user to analyse repositories.

Beyond the tool itself and its applicability, we have discussed the varied results obtained by running it on publicly available repositories. The observed contrast between values obtained for distinct repositories upon analysis of particular parameters (e.g. cohesion & distribution) reveal a number of aspects by which these projects differ. One such aspect is the degree of distributiveness in development: certain projects are manned by only a handful of people responsible for the majority of code written, while other projects adopt a more decentralized approach, with a large number of developers all contributing a little to the codebase.

As it currently stands, our application may already be employed in a wide range of use cases, all of which have been detailed in subsection 5. Currently, we plan on wrapping up the underlying framework under a publicly accessible web service, thus allowing developers and management teams to incorporate its functionalities into their own software projects or research endeavours.

## References

[1] Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. *"Microservices: Yesterday, Today, and Tomorrow." Mazzara M., Meyer B. (eds.), Present and Ulterior Software Engineering. Springer..*

[2] S. Faraj and L. Sproull *"Coordinating expertise in software development teams"*. Manage. Sci., vol. 46, no. 12 Dec. 2000.

[3] J. Herbsleb and R. Grinter *"Architectures, coordination, and distance: Conway's law and beyond"*. Conway's law and beyond," Software, IEEE, vol. 16, no. 5

[4] Teusner, R., Matthies, C. and Giese, P., 2017. *Should I Bug You? Identifying Domain Experts in Software Projects Using Code Complexity Metrics.* 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS),.

[5] Begel, A., Phang, K. and Zimmermann, T., 2010. *Codebook.* Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10,.

[6] Boehm, B. W. (1987). *Industrial software metrics top 10 list.* IEEE Software

[7] Gousios, G., Kalliamvakou, E. and Spinellis, D., 2008. *Measuring developer contribution from software repository data.* Proceedings of the 2008 international workshop on Mining software repositories - MSR '08.

[8] K. Beck, *Extreme Programming Explained: Embrace Change.* Addison- Wesley Professional, 2000.

[9] Hattori, L. and Lanza, M., 2008. *On the nature of commits.* 2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops,.

[10] F. Balmas, F. Bellingrad, F. Denier, S. Ducasse, B. Franchet, J. Laval, K. Mordal-Manet, and P. Vaillergues, "*The squale quality model. mod'ele enrichi d'agr ´egation des pratiques pour java et c++,*" INRIA, Tech. Rep., 2010.

[11] Vasilescu, B., Serebrenik, A. and van den Brand, M., 2011. *You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics.* 2011 27th IEEE International Conference on Software Maintenance (ICSM),.

[12] Kim, S., Whitehead,, E. and Zhang, Y., 2008. *Classifying Software Changes: Clean or Buggy?.* IEEE Transactions on Software Engineering, 34(2), pp.181-196.

[13] Milewicz, R., Pinto, G. and Rodeghero, P., 2019. *Characterizing the Roles of Contributors in Open-Source Scientific Software Projects.* 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR),.

[14] Nordberg, M., 2003. Managing code ownership. IEEE Software, 20(2), pp.26-33.

[15] Marco di Biase, Ayushi Rastogi, Magiel Bruntink, and Arie van Deursen. *The Delta Maintainability Model: measuring maintainability of fine-grained code changes.* IEEE/ACM International Conference on Technical Debt (TechDebt) at ICSE 2019, pp 113-122

[16] GitHub. 2021. reposense/RepoSense. [online] Available at: https://github.com/reposense/RepoSense

[17] GitHub. 2021. bloomar/git-developer-contribution-analysis. [online] Available at: https://github.com/bloombar/git-developer-contribution-analysis

[18] GitHub. 2021. IonicaBizau/git-stats. [online] Available at: https://github.com/IonicaBizau/git-stats

[19] Tomcat. 2021. Apache Software Foundation [online] Available at: https://tomcat.apache.org/tomcat-7.0-doc/developers.html

## List of Figures

## List of Tables