University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Commit Evaluation Engine for an Open Source Donation Platform

Calvin Falter
Zürich, Switzerland
Student ID: 17-708-934

Supervisor: Eder John Scheid, Dr. Thomas Bocek,
Dr. Guilherme Sperb Machado
Date of Submission: November 1, 2020

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Mit dem Wechsel von kommerzieller Software zu Open-Source-Software beteiligen sich immer mehr Entwickler an diesen Projekten. Leider verschwinden die meisten Open-Source-Projekte nach wenigen Tagen. Dafür gibt es mehrere Gründe, wovon einer die mangelnde Finanzierung ist. In dieser Arbeit wird eine Software vorgestellt, mit dem der Beitrag eines bestimmten Entwicklers zu einem Open-Source-Projekt berechnet werden kann. Damit sollen die Spenden für ein Projekt gerecht unter den Mitwirkenden je nach ihrer geleisteten Arbeit verteilt werden können. Zu diesem Zweck werden in bestehenden Studien zur Analyse von Beiträgen verschiedene Metriken evaluiert, weitere mögliche Metriken untersucht und schliesslich eine Auswahl von Metriken getroffen, die auf den Anwendungsfall der Open-Source-Repository-Analyse anwendbar sind. Dies resultiert in Metriken, die vom Versionskontrollsystem Git extrahiert werden können und die durch Informationen ergänzt werden können, welche auf Repository-Plattformen wie GitHub verfügbar sind. Eine Use-Case-Analyse der Software zeigt die Anwendbarkeit der Systems und bestätigt die Korrektheit der berechneten Beiträge.

With the shift from commercial software to open-source software, more and more developers participate in these projects. Unfortunately, most open-source projects disappear after a few days. There are several reasons for this, one of which is the lack of funding. This thesis presents an engine that can be used to calculate an individual developer's contribution to an open-source project. This should enable to distribute the donations for a project fairly among the contributors according to the work they have performed. For this purpose, different metrics are evaluated in existing studies on contributions analysis, further possible metrics are examined and finally, a selection of metrics is made that are applicable to the use case of open source repository analysis. This results in metrics that can be extracted by the version control system Git, which can be supplemented by information available on repository platforms like GitHub. A use case analysis of the software shows the engine's applicability and confirms the correctness of the calculated contributions.

Acknowledgments

First of all, I would like to thank the two initiators of this project, Dr. Thomas Bocek and Dr. Guilherme Sperb Machado. They were and still are enthusiastic about this project and have always supported me with great euphoria. I would also like to express my special thanks to Eder John Scheid, who continuously supported me from the scientific side and always pointed me in the right direction. At this point, I would like to thank Prof. Dr. Burkhard Stiller for letting me be part of an exciting project of the Communication Systems Group of the University of Zurich.

Contents

Chapter 1

Introduction and Motivation

Open source means that a license for a software project allows users to modify and enhance the project. This means that an open-source software can be integrated into other projects and allows reuse of the code and ideas freely. Thus, open-source removes the barriers of collaboration to accelerate innovation and technological advancements. In the last few years, a trend in shifting from commercial software to open-source software was observable [3].

Open-source projects often struggle to keep active, as can be seen in [29], where it was verified that, from 843,763 GitHub projects, most of them only live for 10 days. There are several reasons why most open-source projects fail [5], *e.g.,* due to the lack of engagement or interest from the developer community, usurped by a competitor, lack of time, or obsolescence of the project. However, among all of these reasons, one stands out, which is the lack of funding [6]. The funding of open-source projects is achieved through crowdfunding, paid licenses, sponsorship from big companies, or donations.

These mentioned possibilities to support a project are based on the fact that there is an organization or a structured grouping behind the project. GitHub [16] as the largest open-source software platform has created such a service itself. This service is called GitHub sponsors [15]. Through these sponsors it is possible to support either a developer or an organization. The advantage of this is certainly the direct embedding in the place where the code is located and from where it is often looked at. If you would like to support someone with an amount of money, this can be done as a monthly payment with a predetermined amount. However, it is not possible to simply support a project. Other services like Patreon [39] or Open collective [35] allow this. But how the money is used or shared is up to the owners of the Patreon or Open Collective account or the Git repository.

Therefore, it is impossible to pay the developers for their work on a project with the current funds unless their GitHub sponsors account is found (if it exists). An amount can then be transferred to them. However, this amount is not related to their contribution to this repository. This fact shall now be changed. In a project of which this thesis is a part, a method will be created to support a repository. This is to be done in a way that it is possible to support the repository financially, except the donations do not remain with the organization or the owners of the repositories and have to distribute them manually.

Instead, the donations are distributed to the developers according to the contribution made to the repository. For this purpose, an engine will be developed that analyzes the repository, quantifies the developers' contributions, and calculates a score that reflects individual developers' contributions.

1.1 Description of Work

In an initial stage, this thesis demands to research in the literature, which are the metrics that must be taken into consideration when developing an engine able to quantify the real contribution of a developer towards an open-source project. In this sense, a state-of-the-art review regarding "mining and quantifying developer contribution" must be conducted to gather and classify all the metrics and current works. The expected outcome is a detailed and complete classification and documentation of existing work.

After the relevant metrics are identified in a second stage, the engine needs to be designed in a scalable way. This is important, as large projects such as the Linux kernel have thousands of contributors, which will take time to analyze. The implementation should bring one or more metrics to quantify developer contributions, also covering common use cases. After the design of the engine, it should be implemented. Various test-cases should cover good-path and corner cases as well.

1.2 Thesis Outline

This thesis is divided into six chapters. Chapter 2 gives background information on the topics used in this thesis. Chapter 3 presents various studies that also deal with the quantification of developer contribution using different approaches and metrics. These approaches will be discussed and analyzed, which metrics can be used for this project. Subsequently, Chapter 4 will deal with the commit evaluation engine, which is developed in this thesis's context. First, some requirements are defined. Based on these requirements, the individual metrics will be determined and described how they will be used to calculate the contribution. This chapter also documents the implementation. Chapter 5 then deals with the evaluation of the engine. The performance is tested, and a use-case example is performed. Chapter 6 finishes with a summary and a conclusion.

Chapter 2

Background

As this thesis tackles problems related to version control systems and software quality metrics, it is crucial to describe these areas and related concepts. Thus, Section 2.1 describes the Git version control system, Section 2.2 presents platforms for software repositories, Section 2.3 lists software quality metrics, and, finally, Section 2.4 delves into code complexity calculation.

2.1 Version Control System - Git

Version control systems (VCS) are systems that record and manage changes to one or more files. There are local VCS that record the changes in a database. There exist also centralized VCS. These possess a single centralized database where the changes are recorded. Computers that want to use versions of a file always access this database directly. The last variant is distributed VCS. In these systems, each network member has its personal database on which it manages the versions of the file(s). To work together with other members, it is possible to synchronize the versions and add versions from another network database to the version database [10].

One such system is Git. It is often used in connection with code development and has become the de facto standard. The distributed approach also makes it possible to work collaboratively with other developers. These features are essential for the development of open-source software. Anyone with the necessary rights can participate in a project. Besides, the whole code with the history of the versions is continuously visible to everyone. In most cases, a platform is preferred as a server computer for the version database. The largest such platform is GitHub. However, there are also others, *e.g.,* GitLab and Bitbucket. In theory, it is even possible to self-host such a platform. Retrieving code from a platform works the identical way with the VCS Git, regardless of the platform. There is just the need for a link to a .git file.

In Git, projects that contain version-controlled elements are called repositories. Different branches are possible within a repository. Branches are utilized to have different development lines in parallel. These point to a series of commits, which symbolize the individual

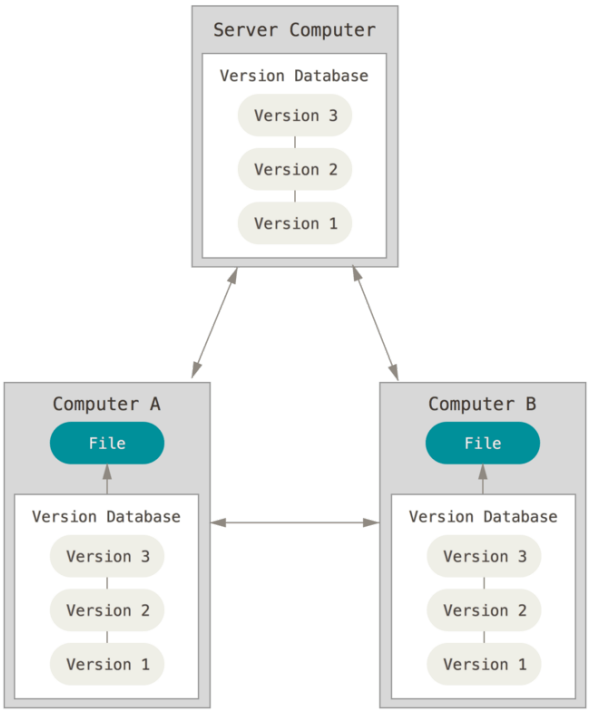

Figure 2.1: Distributed Version Control Systems [10]

versions where changes have been made. A new commit can only be appended to an existing commit in a branch. If several branches exist simultaneously, it is possible to merge one branch's commits into the other branch (*i.e.,* the main branch) so that the main branch now contains all commits of both branches.

2.2 Software Repository Platform

There are many different ways to manage a Git software repository. If collaborative working is desired, it is worth considering a platform with a Git service. Such a platform can either be used by a third party or self-hosted. Known platforms are GitHub [16], GitLab [17] or Bitbucket [1] as third party platforms as well as self-hosted platforms like Gitea [12]. The advantage of a platform with a Git service is that, in most cases, there is more than just an overview of Git activities. It often also serves as a platform for collaborative work.

Issues are a central part of this additional service. They help to record open work. These tasks can be of different nature. In some platforms, it is also possible to assign tags to the issues. These can be assigned as issues to indicate bugs, feature requests, or questions. In an issue, the content of the issue can be discussed with the help of comments.

Pull requests or merge requests, as they are sometimes called, are also part of a Git platform's primary offering. Pull requests allow a planned merge into another branch. They present an overview of the upcoming merge and give the possibility even to run tests on some platforms. A comment function also allows commenting on the planned

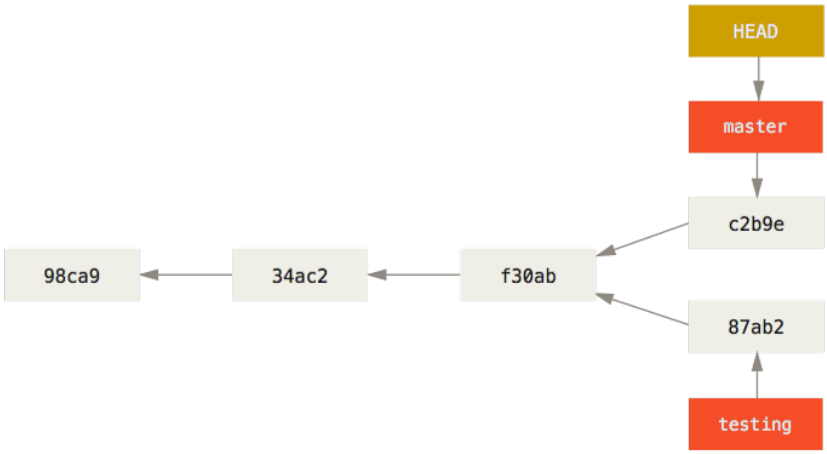

Figure 2.2: Git With Different Branches [11]

merge. On some platforms, this is even implemented in a way that feedback or reviews of the code itself can be made.

2.3 Software Quality Metric

In the past, the ISO standards organization has been involved in determining software quality. For this purpose, they have defined the following six high-level product qualities: functionality, reliability, usability, efficiency, maintainability, and portability [8]. These are accepted by educational experts and academic research [24]. In the revised edition of these standards, these characteristics were defined as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability [25].

For the assessment of the software quality, a tool is often used. In [42], a method was developed to determine precisely this software quality in open source software projects. The subsequent metrics were chosen to make a statement about the characteristics defined in [8].

- Number of statements (counts the average number of executable statements per component)
- Cyclomatic complexity
- Maximum levels (measures the maximum number of nestings in the control structure of a component)
- Number of paths (counts the mean number of non-cyclic paths per component)

- Unconditional jumps (counts the number of occurrences of GOTO)
- Comment frequency (this is defined as the proportion of comment lines to executable statements)
- Vocabulary frequency (defined by [23] as the sum of the number of the unique operands, n1, and operators, n2, that are necessary for the definition of the program.
- Program length (measures the program length as the sum of the number of occurrences of the unique operands and operators)
- Average size (measures the average statement size per component)
- Number of inputs/outputs (counts the number of input and exit nodes of a component)

2.4 Code Complexity

A characteristic to examine and evaluate a code is the code complexity. In scientific work, four methods of assessing code complexity are considered the most popular [44] [2]. These methods include Mc Cabe's cyclomatic complexity [32], Halsted's programming effort [23], statement count as well as Oviedo's data flow complexity [36] [44]. The complexity should help to evaluate the difficulty that programmers have in writing and understanding programs [36]. This should lead to the development of more understandable methods and programs.

Mc Cabe's Cyclomatic Complexity

With his proposal to measure complexity, Mc Cabe wants to address the modularity of programs. Complexity should be independent of the size of programs, which means adding and removing functional statements does not change complexity. He justifies this with the fact that a lot of time and effort at software engineering is invested in testing and maintenance [32]. The degree of modularization should show how difficult a system is to test and maintain. He bases the calculation of complexity on mathematical preliminaries from Berge [4], where n represents the vertices, e the edges, and p the connected components.

$$V(G) = e - n + p \tag{2.1}$$

Mc Cabe applies this formula to the structure of programs and simplifies it to the following.

$$v = \pi + 1 \tag{2.2}$$

Thereby π reflects the number of predicates. This means that the cyclomatic complexity of a structured program equals the number of predicates plus one, whereas compound predicates such as `IF C1 AND C2 THEN` are treated as two predicates as they could be written as `IF C1 THEN IF C2 THEN` [32]. The challenge with automatic detection of complexity would be the recognition of predicates in different languages. Where predicates are detected, it would also be necessary to detect whether they are predicates and how many single predicates they account for.

Halsted's Programming Effort

Halsted also developed a way to measure the complexity of programs [23]. He calls the complexity of a program the programming effort. This is based on his measurement of the volume of a program, which he defines as follows.

$$V = (N_1 + N_2)log_2(\eta_1 + \eta_2) \tag{2.3}$$

where Halsted gives the following meaning to the variables

$$\begin{aligned} N_1 &= \text{Number of distinct operators} \\ N_2 &= \text{Number of distinct operands} \\ \eta_1 &= \text{Total number of operators} \\ \eta_2 &= \text{Total number of operands} \end{aligned}$$

He now defines the programming effort as the ratio of the volume of the program in square to the minimum possible volume of the program V^*.

$$E = \frac{V^2}{V^*} \tag{2.4}$$

Since it is difficult to calculate the minimum possible program volume, the following approximation is often taken [44].

$$E = \frac{\eta_1 N_2 (N_1 + N_2) log_2(\eta_1 + \eta_2)}{2\eta_2} \tag{2.5}$$

Statement Count

This way of determining the complexity of a program is probably the oldest one. A significant advantage of this method is its straightforward approach because it only reflects the number of all program statements. It is still needed to define for each language what a statement is, but then this method is simple to determine the program's complexity. The notable advantage of the statement count as a complexity value is also its disadvantage. The relatively superficial analysis of the count of statements can be seen as not profound enough to determine a program's complexity.

Oviedo's Data Flow Complexity

Two assumptions are made for the data flow complexity of *Oviedo* [44].

1. It is easier to determine the definition-reference connections within a block than between different block and
2. the number of different variables within a block is more important than the total number of all references in a block.

The data flow complexity within a block counts all previous definitions of locally exposed variables in a block i that reach block i. The dataflow complexity of the entire program is then defined as the sum of the complexities of all blocks. This measurement of complexity has a simple approach compared to other complexities because it only considers the definition of the variable and its references. For automatic evaluation of the complexity, this approach has a significant advantage. One difficulty in automatically assessing the complexity of the data flow could be the automatic delimitation of blocks and the recognition of variables and their references in different programming languages.

Chapter 3

Related Work

The central part of this work is finding a method to collect the individual developers' contributions and weight them to finally make a statement about the overall contribution. In the past, this has been tried in several research projects. The main reason why this work could not be based on an already developed method is the application for which the method is used in this context. In the following part, various related works are presented with their application and possible applications to this project.

[30] presents a list of different evaluation metrics. These include code contribution, code complexity, as well as bug-related metrics. In contrast to others who have tried to assess developers' contributions, they try to have the application of these metrics validated by team leaders who also assess the contribution themselves.

The authors of [3] develop in their study visualization possibilities of the contribution behavior in open source repositories. The metrics lines of code (LOC) and number of commits serve as a basis for determining the contribution. The study then builds on these and examines the factors that make up the contribution, patterns in temporal contribution behavior, and the context of the developer's region.

The goal of [38] was to find a metric to evaluate students' project code to quantify the amount of work contributed by each team member. This tool developed in the study is designed to help instructors evaluate each team member fair using quantitative measurements and no longer purely subjectively. They base their quantified contribution on the metrics number of commits, number of merge pull requests, number of files, total LOC, and time spent.

The authors of [21] rely on a variety of data sources to determine the contribution. In addition to the code-specific metrics extracted from the repositories, they also use the sources mailing lists/forums, bug databases, wikis, and chats. The tasks of a developer in this study are not only seen in programming. A developer contributes to a variety of activities that involve both the process and the product. This is the reason for the inclusion of other data sources.

3.1 Discussion

Table 3.1 presents an overview of the related work, of the metrics used, and the use case applied in the study. It can be seen that most of the metrics found in the literature are based on the metric LOC. However, this metric often does not stand alone, as it is considered to be not meaningful enough on its own. The other metrics used are discussed in more detail below.

During the discussion in [30], it became clear that the two contribution metrics of code contribution, which was measured in LOC, and average complexity per method were the most popular among the project leaders. The other metrics, the introduced bugs and bug fixing contribution contained weaknesses, and the leaders realized that it could sometimes punish the wrong developers. To measure complexity, Mc Cabe's Cyclomatic Complexity (Section 2.4) of the added and changed methods was chosen. The presented work is not applicable to the use-case of open-source repositories. Firstly, only Java code is considered in the analysis, and secondly, there are bug databases in every case. For open-source repositories, this is not necessarily the case. What is also seen is that the social aspect is neglected. For future work, it is planned to include data of messenger services and collaboration platforms like GitHub [16] and GitLab [17].

[3] has a different core objective than this work. The focus is on visualizations of the contribution behavior and influencing factors. However, the determination and quantification of the contribution are based on similar assumptions as in this thesis. Both focus on open source repositories and can, therefore, use the same metrics. Hence, building on the metrics and calculations presented in this study is certainly useful. The following equation illustrates the calculation of the contribution with the metrics' commits and LOC.

$$Contri(i) = \frac{LOC(i)}{\sum_{k=1}^{n} LOC(k)} + \frac{Commit(i)}{\sum_{k=1}^{n} Commit(k)}, 1 \leq i \leq n \tag{3.1}$$

n: total number of developers

[38] does not concentrate on open source repositories but has its focus on metrics, which are all available in open source repositories. However, what is noticeable when calculating the contribution is that the work is assumed to be constant, as in a school project. As shown in Table 3.2, the individual values are assigned to a weight. All weights of a contributor are then combined, which ultimately leads to one of the ratings excellent, good, satisfactory, poor, and unsatisfactory. There is, consequently, no continuous numerical evaluation. However, this is something that is needed for this project. Besides, in [38], a measurement of the working time was suggested. This is based on the time patterns of the Git commits. Still, with different commit patterns, it is impossible to tell how much working time is behind a commit for each person. In addition to their Git metrics, they saw that these metrics say not everything about a team member's contribution. To address this challenge, they propose the inclusion of a difficulty gauge to weight the Git metrics. This difficulty gauge is based on the difficulty of the task assigned to the developer. For open source repositories, however, there is no instructor or leader for each project who can quantify

Table 3.1: Comparison of Related Work

Work	Parameters	Use-Case
[30]	• Code Contribution • Average Complexity per Method • Introduced Bugs • Bug Fixing Contribution	In company assessment of contribution to give leaders more statistical information
[3]	• LOC • Number of Commits	Contribution behavior and relationships in open source repositories.
[38]	• Number of Commits • Number of Merge Pull Requests • Number of Files • Total LOC • *time spent*	Objective contribution analysis of school projects to support the instructor rating the participation of the team members
[21]	• Code and documentation repository: Additions / Deletions / Changes / New file / New directory / Bug creation / Bug fixing / Code documentation / Large commits / New documentation file / New translation file / New binary file / Commit message / Reference to bugs • Mailing lists / forums: Reply / New thread / Close lingering thread / Flamewar participation • Bug database: Report / Close / Comment / Close a bug that is reopened • Wiki: New page / Update page / Link to page • Internet Relay Chat: Participation / Replies to directed questions	Full analysis of 48 sub-projects of the GNOME project by history of the source code repositories, mailing list archives and bug reports.

each task's difficulty. Therefore, the approach of considering the difficulty in this way will not be applicable in this context.

Commits / month $(C)^a$	Merges / month $(M)^b$	Files / month $(F)^c$	LOC / month $(L)^d$	Time spent / day $(T)^e$	Weight age
70+	22+	25+	1k+	8+ hrs.	1.0
[60 - 70)	[20 - 22)	[22 - 25)	[0.9k - 1k)	[7.5 - 8)	0.9
[50 - 60)	[18 - 20)	[20 - 22)	[0.8k - 0.9k)	[7 - 7.5)	0.8
[40 - 50)	[15 - 18)	[17 - 20)	[0.7k - 0.8k)	[6.5 - 7)	0.7
[30 - 40)	[12 - 15)	[15 - 17)	[0.6k - 0.7k)	[6 - 6.5)	0.6
[25 - 30)	[10 - 12)	[13 - 15)	[0.5k - 0.6k)	[5 - 6)	0.5
[20 - 25)	[8 - 10)	[10 - 13)	[0.4k - 0.5k)	[4 - 5)	0.4
[15 - 20)	[6 - 8)	[8 - 10)	[0.3k - 0.4k)	[3 - 4)	0.3
[10 - 15)	[4 - 6)	[5 - 8)	[0.2k - 0.3k)	[2 - 3)	0.2
Below 10	Below 4	Below 5	Below 0.2k	Below 2 hrs.	0.1

Table 3.2: Weightage Scheme on Extracted Metric Data from [38]

The contributions analysis of [21] is definitely the most detailed one presented in this chapter. A disadvantage of the many metrics used is that this method is not easily applicable to other projects to be analyzed. In [21], the data basis was provided by projects of the gnome ecosystem, where this data was available.

3.2 Implication

In their current form, all the methods presented are not simply applicable to a contribution analysis engine, so their requirements are still met. In comparison, it becomes clear that the LOC as a metric are present in all studies like a red thread. Therefore, it will be useful to build on these metrics and other Git specific metrics. Other used metrics are not convincing in their consistency of analysis, or they are not applicable because the data is missing in open source repositories or because this analysis is only focused on a single or few selected languages.

Chapter 4

Commit Evaluation Engine

This chapter presents the design of the commit evaluation engine. In Section 4.1 the requirements that drove the engine's development are described. Then Section 4.2 details the full design with employed metrics and architecture. Finally, Section 4.3 presents the implementation of the engine.

4.1 Requirements

The following Requirements (Req) were listed to guide the development of the engine's design and implementation.

Req 1 *Clone Repository - Information Required*: The engine shall be able to clone any publicly available repository just by the provided Git URL. For example, clone the repository from the following URL `https://github.com/go-git/go-git.git`

Req 2 *Clone Repository - Elements*: The engine shall only clone the part of the repository, that will be needed for the analysis. For example, if the analysis shall be conducted on the master branch, only the master branch shall be cloned.

Req 3 *Clone Repository - Frequency*: The engine shall be built in a way that it must clone the repository as rarely as possible but as often as needed.

Req 4 *Update Repository*: The engine shall be built in a way that it can update the repository if it already exists without cloning it again.

Req 5 *Analysis - Repositories*: The engine shall be built in a way that it is capable of analyzing any open source repository independent of chosen stack and languages.

Req 6 *Analysis - Data*: The engine shall only use data from Git itself and optionally from platforms like GitHub [16].

Req 7 *Analysis - Score*: The engine shall provide a score of each contributor for any open source repository. This score shall represent the contribution of the corresponding contributor in relation to the total contribution of all contributors. The score shall be computed in a way that is not exposable. A higher score shall always correspond to a higher contribution.

Req 8 *Analysis - Branch*: The engine shall only analyze one branch. It shall be possible to change the branch to analyze.

Req 9 *Analysis - Time Frame*: The engine shall offer an option to only analyze a repository within a certain time frame. It shall be possible to set an optional lower limit of a starting point from which the repository shall be analyzed. It shall likewise be possible to set an optional upper limit of an endpoint until the repository shall be analyzed.

Req 10 *Analysis - Platform Information*: The engine shall offer an option to optionally analyze the platform information in addition to the default Git analysis.

Req 11 *Architecture - API*: The engine shall be accessible through a REST API with a GET endpoint. Through this API, it shall be possible to deliver the necessary information to analyze the repository according to the configuration of the caller of the endpoint.

Req 12 *Architecture - Setup*: The engine shall be built in a way that some configurations of the backend itself can be made using environment variables.

Req 13 *Architecture - Endpoints*: The engine shall offer two endpoints via the REST API.

- One endpoint shall return the raw contributions per contributor
- One endpoint shall return the calculated score per contributor

Req 14 *Architecture - Parameters - Repository*: The requester of an analysis shall be able to set the repository to analyze via the REST API endpoint.

Req 15 *Architecture - Parameters - Time Frame*: The requester of an analysis shall be able to set the optional time frame for the analysis using a start and an end date via the REST API endpoint.

Req 16 *Architecture - Parameters - Branch*: The requester of an analysis shall be able to set the branch he wants to analyze via the REST API endpoint. If no branch is set, the engine shall pick a default branch set from inside the engine.

Req 17 *Architecture - Parameters - Platform Information*: The requester of an analysis shall be able to set whether he wants to analyze the repository with platform information via the REST API endpoint. The default should be not to integrate the platform information into the analysis.

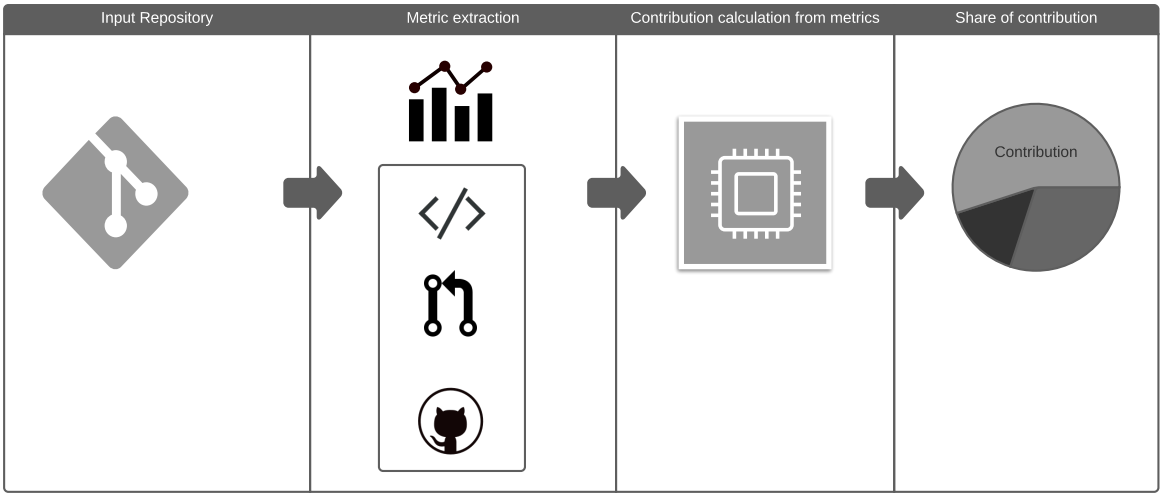

Figure 4.1: Commit Evaluation Engine Flow

4.2 Design

The process of analysis in the engine is structured, as shown in Figure 4.1. The first step is to get all the data needed, which is equivalent to cloning a Git repository. In a second step, the required metrics out of various sources are extracted and collected. In a third step, the summed metrics are offset against each other, resulting in a score for each contributor. Finally, each contributor is given a share of the total contribution.

4.2.1 Metric

Developer contribution is very versatile. The development and existence of a project depend on several factors. Thus, a developer no longer contributes only through written code but also through other activities such as communicating and coordinating with other people involved in the project. [21]

In this paper, we distinguish between two different types of metrics. One is Git and repository-based and contains all metrics that can be read from a repository with Git. Since the analysis engine is based on a Git repository, this Git analysis is possible for all queries. The second source of metrics is platform-specific information. Known platforms that offer Git project management are GitHub [16], GitLab [17], Bitbucket [1] and others. All these platforms differ in their API. For each platform, the analysis must be implemented again. Therefore it is not possible to include metrics of platform-specific data for each repository to be analyzed. For the sake of completeness, in this section, we will also have a look at code-specific metrics. However, we will not use these metrics for the engine.

Git-specific metrics

Git is originally very decentralized. With the introduction of platforms, there has been a more centralized approach to code storage and management. The approach of mutual pulling code from other developers in a team, as Linus Torvalds once described it, has receded into the background [43]. When cloning a repository or pulling from a different source, only the version in the Git database will be updated. The individual Git interactions of developers that take place on their own devices, such as switching a branch, will not be shared with other sources. As an external party that clones the source code on GitHub, limited access to the history of actions is available. With the reflog command, it is possible to find most of the Git interactions, but these are only accessible on the device on which the interaction was made. The following metrics are those, which are available on all devices that clone the repository.

Git is structured in its version control system so that the smallest unit of change is a commit. To contribute to a repository as a Git user, he must commit his changes. The metric of the number of commits per person is therefore needed to calculate a contribution to a project.

Merges also count as commits. The only difference between commits and merges is that a merge inserts the existing commits from one branch into another branch. There are thus two input branches and only one output branch. A merge does not necessarily contain a contribution but only helps to gather the contributions and resolve any conflicts. Branches, however, are an essential part of the version control system Git, which is why merges are also necessary to arrive at an end product. In this analysis, merges must, therefore, be taken into account.

Commits can vary greatly in size. It is up to each developer to decide how many changes they want to include in a commit. Each commit contains certain changes, which are reflected in LOC added and removed. As Kan mentioned in 2003, the measurement of a person's contribution is either possible through LOC or function points [28]. In previous studies that tried to measure the contributions of a developer, the LOC were chosen as the main metric [30] [21]. These commits consist of the additions and deletions. In this study, additions and deletions are looked at separately and not as a difference (e.g., +12 -8 is not viewed as a contribution of +4) since this would allow for negative contributions, which can happen during refactoring. This would encourage contributors simply to write unnecessary long code since this would be valued more than short code. Rewriting code in a more compact way would also count as negative contribution, which it is certainly not.

The only time it is known that a commit surely does not end in a contribution is when a revert is applied to that commit. Git offers this revert option, but it is no different from a commit that inverts the last commit changes. Git suggests `Revert "last commit message"` as the default commit message. Reverts would be found according to this naming pattern. However, the commit messages can be freely customized, so it is no longer possible to search for the commit message ”revert” with a search function and find all reverted commits with certainty. Therefore this metric would only apply to some cases

where the message is not changed and would allow for a bypass or even for faked reverts. Since this would not be a fair evaluation scheme, it will not be used in this study.

Dangling commits also occur when a change is undone. The difference between reverts, where a new commit is added, and the process of dangling commits is that the status of an earlier commit is pushed as the new head. The commits that would have come after this new head commit are thus truncated. There are also no longer references to these objects. Therefore, they are generally removed from the Git garbage collector. In theory, however, it is possible to retrieve these dangling commits if they have not yet been removed. However, this requires Git commands, which can only be executed on the computer the commits were truncated. Since the analysis engine clones the repository, these dangling commits are not recorded. An execution of these commands on the servers from which the repository was cloned would have been necessary. Since this is not possible, this metric is not used in this analysis engine.

Squashes are needed when a combination of several small commits into one larger commit is desired. From one point of view, squashes may reduce the complexity of countless commits on different branches with merges and other actions, but with squashes also the power of some features like `git blame` or `git bisect` is reduced [31]. In the end, it is not generally possible to say whether a squash makes sense and that it brings the project closer to its goal. It is quite possible that a squash is counterproductive for the VCS. It must therefore be considered subjectively from case to case. Since case-by-case consideration in an analysis engine is not possible, and the judgments must not be subjective, squashes are not considered.

Analysis on code-level

Another possibility to further analyze the contribution is to go into the commit's content and analyze and consider it as a metric. As [22] has already shown, it can be useful to evaluate the content, punish created bugs, and rate commits that fix bugs better. However, identifying bugs and finding the corresponding fixing is a challenge. The detection of bugs just by going through the lines of code is very limited. Integrated development environments (IDE) already support the detection of certain bugs in a similar fashion, but the detected bugs are not very complex, but mostly just syntax errors. Parsing the code to detect syntax errors alone requires a separate parsing method for each language. Detecting bugs that are more complex than syntax errors requires a different approach. One possibility is to maintain a bug database like it was done in [22]. During analysis, a commit can then be checked against this database to see if a bug was introduced or fixed with that commit. Another way to detect bugs is by running tests. However, these tests must be written solidly so that they will detect the bugs in any case. So testing software requires the execution of scripts. For this purpose, however, a configuration must be made. This conflicts with the approach of a fully automatic evaluation using only the repository address. Additionally, it would allow someone to execute malicious code in the analysis engine.

Another way to analyze a commit specifically concerning Git is to examine the code for conflicts. Merge conflicts can occur when Git cannot merge two existing branches on its

own. In such a case, Git merges the changes it can merge, and the other code snippets Git cannot merge are inserted below each other, with both branches declared as such in the code. The user must then resolve these conflicts on his own. However, it should be rare that someone pushes changes with conflicts because development environments always highlight such conflicts, and GUIs for Git warn to push the changes that way. Searching for such conflicts in an open-source repository is, in almost all cases, useless, as they simply do not exist. Therefore, taking this metric into account would only increase the complexity and runtime of the repository analysis and would hardly bring any advantage.

Fundamentally, this analysis at the code level is one to check the code quality. The above examples were simple executions of code quality determination. Another more advanced way to determine code quality is via code complexity.

As described in the Chapter 2, there are four common methods to determine a code's complexity. A common feature of all methods is that they require the execution of a script. Besides, all of the presented methods require an interpretation of the code to determine its complexity. In order to apply such a method, the determination and interpretation of a language are essential. Nonetheless, since Req 5 says that the engine should be able to analyze all public repositories, it should be possible to determine all languages of all repositories. There is also the difficulty that often several file types and languages are used in one repository. The determination would be even more difficult if different languages were used within a file. Even if this could be determined, one would have to decide on an adequate method and apply it to all languages. This can be very tricky because the complexity of HTML code is challenging to compare with Java code.

Exactly this difficulty seems to have been encountered by different already available tools. These often only support a certain number of selected languages in the analysis of complexity. This is also the case with general tools for determining code quality. A simple automatic analysis without first configuring this code analysis engine is not possible. One reason for this configuration is that with external tools, the execution of tests is part of the analysis process. This is not compatible with the automatic execution as provided for in the contribution analysis engine. Another argument against such tools is the performance impact on the engine. As soon as tests have to be executed, the analysis takes longer again. In our case, it is also desirable to trace the code quality back to different contributors. This requires multiple executions of determining the code quality for each version since these tools do not list the quality by contributor but by project or file or function. However, since even a function can be written by several people, this assignment to contributors is only possible if the code quality is searched for after each commit after a change of the code quality.

Platform-specific metrics

In modern software development and especially open-source software development, a software developer must now do more than just develop software. Writing code is now only one activity besides communication, coordination, documentation, and many more [21]. A team that develops open-source software and only plans to code will quickly reach its limits. Some of the other activities that an open-source software developer undertakes are

reflected in publicly available information on platforms such as GitLab [17], GitHub [16] or Bitbucket [1].

Unfortunately, these platforms are not uniform, especially in their API. Also, not every platform on which a Git repository is hosted has the same functionality that reflects this additional activity. The central elements offered by a Git service platform include issues and pull requests. If information is to be taken into account in the analysis platform, it is important that it does not differ greatly from platform to platform. For this reason, this analysis engine uses information that is also available on platforms that will be integrated at a later point in time. GitHub was chosen as the first platform. The reason for this choice is the size of GitHub. With over 40 million users and over 100 million repositories, it is the largest platform providing a Git service [14, 27, 16].

The choice of metrics to determine the contribution based on platform information depends on two factors. First, the metrics must be related to the contribution, *i.e.,* they must provide a benefit for achieving a project goal. Due to the diversity of the different platforms that will be supported by the system in the future, it must be ensured that the evaluation of platform information does not essentially differ. A different evaluation of the contribution depending on which platform the repository is hosted on contradicts the neutral analysis of the repository. For this reason, only information based on the core functionalities of a platform and thus represented by almost all larger platforms are chosen as platform information metrics.

A central element of a Git platform is an issue tracking system. Issues can be used to keep track of bugs, enhancements, or other requests [13]. Issues are, consequently, very versatile. A common feature of all possible applications is that an issue reflects work that is still open. In case of a bug, it is a software malfunction that needs to be fixed. If it is an enhancement, it requests an extension or improvement of the existing software. However, not every issue brings the project closer to its goal. Since anyone can create an issue, issues may be created with no or low priority for implementation. In general, issues with much discussion can be considered relevant because several people have dealt with it. To give an opinion on a comment without writing a new comment, GitHub, for example, allows responding to a comment with reactions. Often the thumbs-up emoji is used to enhance the importance of a comment. GitHub offers even more options for issues. For example, tags can be assigned, or persons can be assigned to an issue. In this thesis, however, the metrics of creating issues and adding comments are considered. These metrics contribute to a discussion of work to be done, which brings the project closer to its goal, as errors and opportunities for improvement can be identified and features requested by different people can be weighted and inserted into a backlog.

The other central element of a Git platform are the pull requests or merge requests, as they are called on some platforms. These allow a planned and structured merge to be performed. A branch is selected to be merged into another one. The platform then offers various overviews of this upcoming merge. For example, it can be seen which commits will be added, which changes will be made at the code-level or whether merge conflicts will occur. Furthermore, depending on the repository configuration, the branch can be tested for correctness by executing automated tests. In addition to the automatic possibility of having the code checked, these pull requests also allow for a code review

from other developers. In these code reviews, which can also be requested by the pull request's creator, the developers who submit a code review can add various comments. Comments, suggestions for improvement, or requests for changes can be made down to one line of code, which the creator of the pull request can then respond to. If a code reviewer is satisfied with the code to be merged, he can approve the pull request. All these functionalities of a pull request allow inserting code changes that have been checked by several people in the existing code of another branch. In this thesis, not all possibilities of pull requests are considered because they are only offered on some platforms or since the analysis of the metric would take too long, as can be seen in the example of executing tests. Therefore only the creation of a pull request, the state of the request, and code reviews are considered as metrics in the category of pull requests.

4.2.2 Developer Contribution

The developer contribution should consist of the metrics mentioned in the previous chapter. The goal is to use the existing metrics, which are very limited due to availability or other reasons, to create a formula that precisely reflects the actual contribution of a developer. For this reason, a formula has been developed, which was discussed with open-source contributors, to compare their estimation of the reflection of the real contribution to the calculated one.

As mentioned in the previous chapter, the following metrics have been defined as the ones to be used. The total number of metrics to be used is divided into two groups, with only the first (Git-specific metrics) used in each analysis and the second (platform-specific metrics) only for those where this is desired.

Git-specific metrics:

- **Changes**
  - Additions
  - Deletions
- **History**
  - Commits
  - Merges

Platform-specific metrics:

- **Issues**
  - Author of Issues
  - Comments on Created Issue
  - Written Comments

- **Pull Request**
  - Author of Pull Request
  - Activity on Created Pull Request
  - Performed Code Reviews

All metrics are subordinated to categories in the above list. This subdivision and allocation to categories allow a better overview of the weighting of the individual metrics. The weighting of the individual metrics is done in two stages. In the first stage, the weighting within a category is defined, and in the second stage, the weighting among all categories. A weight within a stage is always defined so that all weights can be added up to 1.

The contribution is then calculated in two stages in accordance with the defined weightings. In the first step, the developer's contribution to be analyzed is calculated as a percentage of the category's total contribution. In the second step, these calculated percentage contributions of a category are balanced against the categories' previously defined weightings. This results in the following formula for the contribution of a contributor.

$$C_{developer} = \sum_{c=0}^{c_{total}} \omega_c \times \frac{\sum_{m_c=0}^{m_{c,total}} \omega_{m_c} \times \theta_{m_c}}{\sum_{m_c=0}^{m_{c,total}} \omega_{m_c} \times \Theta m_c} \tag{4.1}$$

where $C_{developer}$ = Calculated contribution of the developer
c_{total} = Total amount of categories
ω_c = Weight of the category c
c = Index of the category
m_c = Index of the metric inside category c
$m_{c,total}$ = Total amount of metrics within category c
ω_{m_c} = Weight of the metric inside category c
θ_{m_c} = Value of the metric for the analyzed developer
Θ_{m_c} = Total value of the metric for all developers

Analyzing the repository for each developer's contributions using Equation 4.1 will result in a list of each contributor with their percentage of the total contribution to the project within the analyzed time frame. The following tables provide an overview of the categories used as well as metrics with the corresponding weights. The weightings were developed in a process in which they were initially determined based on [38] and self-assessment. They were adjusted through applications in real projects with the agreement of open-source contributors. Table 4.1 shows the distribution of the weights if no platform information is to be analyzed and Table 4.2 shows the distribution of the weights if an analysis of the platform information is made.

ω_c	**Category**	ω_{m_c}	**Metric**
0.55	changes		
		0.7	additions
		0.3	deletions
0.45	history		
		0.7	commits
		0.3	merges

Table 4.1: Weights for Analysis Without Platform Information

ω_c	**Category**	ω_{m_c}	**Metric**
0.36	changes		
		0.7	additions
		0.3	deletions
0.30	history		
		0.7	commits
		0.3	merges
0.14	issues		
		0.5	author of issues
		0.2	comments on created issue
		0.3	written comments
0.20	pull requests		
		0.7	author of pull request
		0.3	performed code reviews

Table 4.2: Weights for Analysis With Platform Information

Additionally to this metric weighting, there are also different weights of the pull requests due to the activity on a pull request. A closed request counts 0.6; an open one counts as one. If a pull request is merged, it counts 1.5 times. In addition to these states, the activity of a pull request is also checked for approval of the request. If this is the case, the value mentioned above is multiplied by 1.4. For example, an approved request, which was closed again afterwards, receives a weight of 0.84 and an approved and merged request receives a weight of 2.1. This multiplier is intended to promote agreement among the developers and to assure the code quality since there is no other representation of code quality in the evaluation.

4.2.3 Architecture

Language

As a backend language for the analysis engine, the language Golang was chosen. It is characterized by its efficient and straightforward code. The still relatively young language

from 2009, which has released the first stable version in 2012, has excellent potential. Go's advantages, which are also used for this project, are the direct compilation to machine code, which makes programs written with Go very fast and, as a second advantage, the possibility of executing different methods concurrently using go routines. Especially in a case where data is collected from different sources, sometimes different parts can be executed simultaneously, and the results can be recombined with channels.

Git interactor

Since the engine is based on data and repositories available through Git, it is necessary to interact with Git. This is the only way to clone repositories and read data from them. Since Golang was chosen as language, a library for Go that meets these requirements was obvious. The recommendation of Git is to use go-git [18]. One advantage of this library is that it does not have any native dependencies. It is as well transparent to the standard Golang performance analysis tooling like CPU, memory profilers, etc. [9].

Server

As an HTTP server for letting the users contact the backend via a REST API, the gorilla/mux library [20] is chosen. It allows for the simple use cases it is needed for in this project, like serving an HTTP request multiplexer, routing the requests to the correct controller in the backend, and returning the computed results.

Activity Flow

The activities related to the analysis of a repository are shown in Figure 4.2. To analyze a repository, a user makes a request to the analysis engine. The analysis engine then initiates two processes.

The first process takes care of platform information. It looks at whether the platform information should be included in the analysis and, if so, whether the analysis engine supports the platform. The desired information, namely the issues and pull requests, is then requested and filtered by the platform. Filtering is done so that the analysis can be performed within a certain period, as defined in Req 9.

The second one of these processes deals with the acquisition of the information available from Git. The first step is to ensure that the analysis is performed on an up-to-date repository. To do this, it is either cloned or updated. The next step is to loop through the commits and extract the defined metrics.

Once both of these processes are completed, the collected platform information is appended to the Git information per Git user. The result is a list of all observed metrics per Git user. Then the formula defined in Equation 4.1 is applied to calculate the contribution. This contribution per Git user of the total contribution is finally returned as a response to the user that made the request.

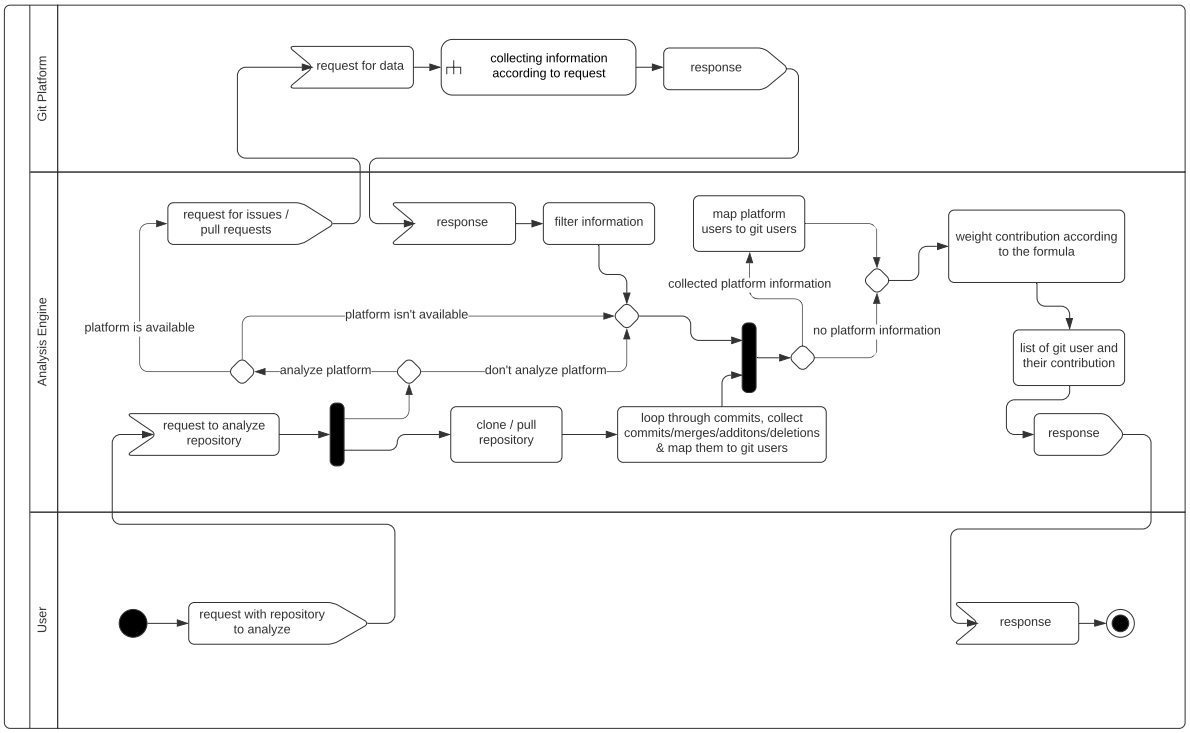

Figure 4.2: Activity Flow Analysis Engine

4.3 Implementation

4.3.1 Cloning and Updating the Repository

As Req 2 defined, the analysis engine shall only clone the part of the repository that is necessary for the analysis. Therefore, the cloning command is executed in a way that it will clone the repository only with the branch to analyze. This branch is defined as the one that is either set via the request or the default one from the environment variables (Listing 4.1).

```
1  func getBranchToAnalyze(r *http.Request) string {
2      branchUrlParam := r.URL.Query()["branch"]
3      // check whether the param was set. If it was
4      // return this branch name, else return the default one
5      if len(branchUrlParam) > 0 {
6          return branchUrlParam[0]
7      } else {
8          return os.Getenv("GO_GIT_DEFAULT_BRANCH")
9      }
10 }
```

Listing 4.1: Detection of the Branch to Analyze

The engine will first try to update the repository. If this process fails because of the error that the repository does not exist yet, the engine will clone the repository. The cloning will be executed so that the cloned repository will only contain one branch, which is the

one defined earlier. To achieve this effect, the repository will be configured as a single branch repository clone with the specific branch as this single branch (Listing 4.2). This means only the history leading to the tip of a single branch is cloned. Further fetches into the resulting repository will only update the remote-tracking branch for the branch this option was used for the initial cloning. In go-git, there exist two possibilities to clone a repository. Either it will be cloned into memory or to the disk in a defined directory. The analysis engine will follow the second approach as this will fulfill Req 3, that a repository shall be cloned as rarely as possible. If, in the future, when the repository is deployed, only a limited amount of space on the disk is left, a procedure for the cleanup of unused repositories must be developed. However, since only a part of the repositories is cloned, this issue can be postponed for future work. The directory structure to the cloned repositories will be built analogous to the structure of the GOPATH in the go environment. The GOPATH environment variable is used to specify directories outside of $GOROOT that contain the source for Go projects and their binaries [19].

Example of the directory structure:

- `https://github.com/go-git/go-git.git`
  ↳ BASE_PATH/github.com/go-git/go-git.
- `https://gitlab.com/fdroid/fdroidclient.git`
  ↳ BASE_PATH/gitlab.com/fdroid/fdroidclient.

```
1  func CloneRepository(src string, branch string) (*git.Repository, error) {
2      folderName := src[8 : len(src)-4]
3      // clone just one branch
4      return git.PlainClone(os.Getenv("GO_GIT_BASE_PATH")+"/"+folderName,
           ↪ false, &git.CloneOptions{
5          URL:           src,
6          Progress:      os.Stdout,
7          ReferenceName: plumbing.ReferenceName(fmt.Sprintf("refs/heads/%s",
               ↪ branch)),
8          SingleBranch:  true,
9      })
10 }
```

Listing 4.2: Cloning the Repository as a Single Page Repository

If the repository already exists, the repository will only be fetched. This supports Req 3 to clone the repository as little as possible. Therefore, once the repository is cloned, it is only fetched so that Git will update the remote changes in the local repository. However, Req 16 makes this process more difficult because, in a request, a different branch was selected for analysis than the one that has initially been cloned with the configuration of the single-branch repository. Therefore, a simple switch of the branch is not possible since the new branch may not even exist in the local repository. In this engine, this case is solved by locally creating a new branch to which the corresponding remote branch's commit history is mapped (Line 10, Listing 4.3). If the branch is already present locally, the Git will append the new commits to this existing branch if there are changes on the remote. After updating the repository, the engine will checkout the selected branch to analyze the correct one (Line 26, Listing 4.3).

```
1  func UpdateRepository(src string, branch string) (*git.Repository, error) {
2      folderName := src[8 : len(src)-4]
3      repo, err := git.PlainOpen(os.Getenv("GO_GIT_BASE_PATH") + "/" +
           ↪ folderName)
4      if err != nil {
5          return nil, err
6      }
7      // take just one remote branch and assign it to a local branch with the
           ↪  same name
8      firstRefSpecArgument := fmt.Sprintf("refs/heads/%s:refs/heads/%s",
           ↪ branch, branch)
9      err = repo.Fetch(&git.FetchOptions{
10         RemoteName: "origin",
11         RefSpecs:   []config.RefSpec{config.RefSpec(firstRefSpecArgument)},
12     })
13     if err != nil && err.Error() != "already up-to-date" {
14         return repo, err
15     }
16     w, err := repo.Worktree()
17
18     if err != nil {
19         return repo, err
20     }
21
22     // checkout the correct branch
23     err = w.Checkout(&git.CheckoutOptions{
24         Branch: plumbing.ReferenceName(
25                     fmt.Sprintf("refs/heads/%s", branch)
26                 ),
27         Force:  true,
28     })
29
30     return repo, nil
31 }
```

Listing 4.3: Updating the Repository

4.3.2 Git Analysis

Once the engine has cloned or updated the repository to be analyzed, the time range to be analyzed is defined (Line 4-10, Listing 4.4). This time range is optionally provided as a route parameter in the request. If not, the API uses the zero values of time.time for the values since and until, which determine the time range. The defined time limits are now passed to the Git log command. As the output of the Git log command, a list of all commits of this branch is accessible. These commits are now looped through to extract the desired metrics.

In this process, a map is built, containing the Git contributor as key and the summed contribution metrics for this user as value. The commits are distinguished from merges by the number of parent commit hashes. Commits have only one parent commit, whereas merges have two (Line 23, Listing 4.4). The other Git metrics that are extracted are the

additions and deletions. These are called with the stats command for each commit (Line 28, Listing 4.4). The additions and deletions also distinguish between commit and merge. Additions and deletions of a merge are calculated with a different factor than those of a normal commit (Line 44, Listing 4.4). This is because in the stats of a merge, all changes made to the commits on this branch before the merge are listed again. So this corresponds to the size of a merge. Hence, the size is still considered because a larger merge also means more work, for example, reviewing.

```
1  var timeZeroValue time.Time
2  var options git.LogOptions
3
4  if since != timeZeroValue {
5      options.Since = &since
6  }
7
8  if until != timeZeroValue {
9      options.Until = &until
10 }
11
12 commits, err := repo.Log(&options)
13
14 err = commits.ForEach(func(c *object.Commit) error {
15     author := Contributor{
16         Name:  c.Author.Name,
17         Email: c.Author.Email,
18     }
19
20     merge := 0
21     commit := 1
22
23     if len(c.ParentHashes) > 1 {
24         merge = 1
25         commit = 0
26     }
27
28     stats, err := c.Stats()
29     if err != nil {
30         return err
31     }
32     changes := CommitChange{
33         Addition: 0,
34         Deletion: 0,
35     }
36
37     // count the lines if its not a merge, otherwise use a factor
38     if merge == 0 {
39         for index := range stats {
40             changes.Addition += stats[index].Addition
41             changes.Deletion += stats[index].Deletion
42         }
43     } else {
44         for index := range stats {
45             changes.Addition += int(float64(stats[index].Addition) *
                   ↪ mergedLinesWeight)
46             changes.Deletion += int(float64(stats[index].Deletion) *
                   ↪ mergedLinesWeight)
```

```
47          }
48      }
49
50      if _, found := authorMap[author]; !found {
51          authorMap[author] = Contribution{
52              Contributor: author,
53              Changes:     changes,
54              Merges:      merge,
55              Commits:     commit,
56          }
57      } else {
58          authorMap[author] = Contribution{
59              Contributor: author,
60              Changes: CommitChange{
61                  Addition: authorMap[author].Changes.Addition + changes.
                        ↪ Addition,
62                  Deletion: authorMap[author].Changes.Deletion + changes.
                        ↪ Deletion,
63              },
64              Merges:  authorMap[author].Merges + merge,
65              Commits: authorMap[author].Commits + commit,
66          }
67      }
68      return nil
69  })
```

Listing 4.4: Git Analysis

4.3.3 Platform Information From GitHub

Depending on the configuration in the request, the analysis engine also supports the inclusion of platform information. As the first platform, only GitHub is currently supported. With its GraphQL API, it is possible to retrieve only the information needed. All issues and pull requests with their associated information are collected in multiple requests. In this implementation explanation, only the process of data collection for the issues is explained. The process of obtaining the pull request data works analogously.

```
1  var timeZeroValue time.Time
2
3  sinceFilterBy := ""
4  pageLength := 100
5
6  if since != timeZeroValue {
7      sinceFilterBy = `since: "` + since.Format(time.RFC3339) + `"`
8  }
9
10 query := fmt.Sprintf(
11     `{
12         repository(owner:"%s", name:"%s") {
13             issues(first:%d, filterBy: {%s}) {
14                 pageInfo {
15                     endCursor
16                     hasNextPage
```

```
                }
                nodes {
                    title
                    number
                    author {
                        login
                    }
                    comments(first: %d) {
                        pageInfo {
                            endCursor
                            hasNextPage
                        }
                        nodes {
                            author {
                                login
                            }
                            updatedAt
                        }
                    }
                    updatedAt
                }
            }
        }
    }`,
    repositoryOwner, repositoryName, pageLength, sinceFilterBy, pageLength)

resp, err := manualGithubGQL(query)
if err != nil {
    return []GQLIssue{}, err
}
var response RequestGQLRepositoryInformation
if err := json.Unmarshal(resp, &response); err != nil {
    return []GQLIssue{}, err
}
```

Listing 4.5: GitHub Initial Issues Request

The data received from GraphQL request is now checked whether all data has been sent or whether further data must be collected in additional requests. GitHub only returns all data in pages with a maximum length of 100 entries. So if there are more than 100 issues or issue comments, there is to need to fetch them in multiple requests. If more than 100 issues or issue comments are present, GitHub will set the entry "hasNextPage" to true and use "endCursor" to provide a pointer to the next page's beginning. In this case, an additional fetch is executed. In Listing 4.6, data is fetched and added to the already fetched data. This is repeated as long as GitHub indicates a next page, and, hence, not all data has been sent yet.

```
issuesAfter := ""

for ok0 := true; ok0; ok0 = response.Data.Repository.Issues.PageInfo.
    ↪ HasNextPage {
    if response.Data.Repository.Issues.PageInfo.HasNextPage {
        issuesAfter = response.Data.Repository.Issues.PageInfo.EndCursor
        issueRefetchQuery := fmt.Sprintf(
            `{
```

```
8               repository(owner:"%s", name:"%s") {
9                   issues(first:%d, filterBy: {%s}, after: "%s") {
10                      pageInfo {
11                          endCursor
12                          hasNextPage
13                      }
14                      nodes {
15                          title
16                          number
17                          author {
18                              login
19                          }
20                          comments(first: %d) {
21                              pageInfo {
22                                  endCursor
23                                  hasNextPage
24                              }
25                              nodes {
26                                  author {
27                                      login
28                                  }
29                                  updatedAt
30                              }
31                          }
32                          updatedAt
33                      }
34                  }
35              }
36          }`, repositoryOwner, repositoryName, pageLength, sinceFilterBy,
                ↪ issuesAfter, pageLength)
37          resp, err := GClientWrapper.Query(issueRefetchQuery)
38          if err != nil {
39              return response, err
40          }
41          var refetchResponse RequestGQLRepositoryInformation
42          if err := json.Unmarshal(resp, &refetchResponse); err != nil {
43              return response, err
44          }
45          response.Data.Repository.Issues.Nodes = append(response.Data.
                ↪ Repository.Issues.Nodes, refetchResponse.Data.Repository.
                ↪ Issues.Nodes...)
46          response.Data.Repository.Issues.PageInfo = refetchResponse.Data.
                ↪ Repository.Issues.PageInfo
47      }
48  }
49
50  return response, nil
```

Listing 4.6: GitHub Issues Pagination

The approach to fetch the missing issue comments works similarly. The engine loops through all issues and checks if there is more to fetch from the corresponding issue comments. If so, the issue comments are fetched for this issue and appended to the previously collected data until the parameter "hasNextPage" of the specific issues issue comments is false.

```
issueCommentsAfter := ""
var issueToRefech int

for index := range response.Data.Repository.Issues.Nodes {
    issueToRefech = response.Data.Repository.Issues.Nodes[index].Number
    for ok1 := true; ok1; ok1 = response.Data.Repository.Issues.Nodes[index
        ↪ ].Comments.PageInfo.HasNextPage {
     if response.Data.Repository.Issues.Nodes[index].Comments.PageInfo.
        ↪ HasNextPage {
      issueCommentsAfter = response.Data.Repository.Issues.Nodes[
          ↪ index].Comments.PageInfo.EndCursor
      specificIssueQuery := fmt.Sprintf(
          `{
          repository(owner:"%s", name:"%s") {
              issue(number: %d) {
                  title
                  number
                  author {
                      login
                  }
                  comments(first: %d, after: "%s") {
                      pageInfo {
                          endCursor
                          hasNextPage
                      }
                      nodes {
                          author {
                              login
                          }
                          updatedAt
                      }
                  }
                  updatedAt
              }
          }
      }`, repositoryOwner, repositoryName, issueToRefech, pageLength,
          ↪  issueCommentsAfter)
      resp, err := GClientWrapper.Query(specificIssueQuery)
      if err != nil {
          return response, nil
      }
      var refetchResponse RequestGQLRepositoryInformation
      if err := json.Unmarshal(resp, &refetchResponse); err != nil {
          return response, nil
      }
      response.Data.Repository.Issues.Nodes[index].Comments.Nodes =
          ↪ append(response.Data.Repository.Issues.Nodes[index].
          ↪ Comments.Nodes, refetchResponse.Data.Repository.Issue.
          ↪ Comments.Nodes...)
      response.Data.Repository.Issues.Nodes[index].Comments.PageInfo
          ↪ = refetchResponse.Data.Repository.Issue.Comments.PageInfo
     }
    }
}

return response, nil
```

Listing 4.7: GitHub Issue Comments Pagination

After all data has been collected, the data is additionally filtered. GitHub offers only one filter since, so there is the need to loop through all issues and issue comments again to check whether they belong to the investigated time frame.

```
1 var timeZeroValue time.Time
2 var filteredIssues []GQLIssue
3 for index := range issueEdges {
4     if (issueEdges[index].UpdatedAt.After(since) || since == timeZeroValue)
      ↪  && (issueEdges[index].UpdatedAt.Before(until) || until ==
      ↪ timeZeroValue) {
5         filteredIssues = append(filteredIssues, issueEdges[index])
6     }
7 }
8 return filteredIssues
```

Listing 4.8: GitHub Filter Issues

```
1 var timeZeroValue time.Time
2 var filteredIssueComments []GQLIssueComment
3 for index := range comments {
4     if (comments[index].UpdatedAt.After(since) || since == timeZeroValue)
      ↪ && (comments[index].UpdatedAt.Before(until) || until ==
      ↪ timeZeroValue) {
5         filteredIssueComments = append(filteredIssueComments, comments[
          ↪ index])
6     }
7 }
8 return filteredIssueComments
```

Listing 4.9: GitHub Filter Issue Comments

Since GitHub only assigns this data to GitHub users and not to Git users, there is the need to map the GitHub user data to the Git users that have already measured their Git contribution. However, it is not possible to read the corresponding Git user to a GitHub user. The mapping in this engine works as in Listing 4.10. In a request, the engine looks in a specific repository for a commit made by a Git user with the same email address as the Git user for which the corresponding GitHub user is being looked for (Line 7, Listing 4.10). With this mapping, the platform information collected about the Git user can be added to the Git contributions. However, be aware that the collected platform information for a GitHub user is only added to a single Git user since a GitHub user can manage multiple Git users.

```
1 query := fmt.Sprintf(
2     `{
3         repository(owner:"%s", name:"%s") {
4             ref(qualifiedName: "master") {
5                 target {
6                     ... on Commit {
7                         history(first: 1, author: {emails: "%s"} ) {
8                             nodes {
9                                 author {
```

```
10                                          name
11                                          email
12                                          date
13                                          user {
14                                              login
15                                          }
16                                      }
17                                  }
18                              }
19                          }
20                      }
21                  }
22              }
23      }`, repositoryOwner, repositoryName, email)
24
25  resp, err := GClientWrapper.Query(query)
26  if err != nil {
27      return "", err
28  }
29  var response RequestGQLRepositoryInformation
30  if err := json.Unmarshal(resp, &response); err != nil {
31      return "", err
32  }
33  if len(response.Data.Repository.Ref.Target.History.Nodes) < 1 {
34      return "", errors.New("could not find user")
35  }
36  return response.Data.Repository.Ref.Target.History.Nodes[0].Author.User.
    ↪ Login, nil
```

Listing 4.10: GitHub User to Git User Mapping

4.3.4 Concurrency

Another requirement defined for the thesis in Section 1.1 is that the system must be scalable. A central feature of Go is the support of concurrency with so-called go routines. Many processes in the analysis engine depend on a sequential sequence. However, there are two time-consuming sequences that can be executed in parallel. The repository cloning and updating as well as collecting the Git metrics (Line 11, Listing 4.11) can be executed in parallel to collecting the platform information (Line 22, Listing 4.11). However, since the platform information does not have to be included in each case, this go routine is only executed if needed. If this is not the case, the channel on which the data would be returned by the routine will be closed immediately (Line 6, Listing 4.11), and the results on the channel will not be waited for (Line 35, Listing 4.11). On Line 43, Listing 4.11, the results of the channels of the two go routines are listened to. These are then assigned to the already initialized variables. With these variables updated, the analysis engine continues in a single sequence.

```
1  // make the channels for both go routines (analyze repo / platform
   ↪ information)
2  gitAnalyzationChannel := make(chan GitAnalyzationChannel)
3  platformInformationChannel := make(chan PlatformInformationChannel)
4
```

```
5  // if we don't have to analyze the platform, close the channel again since
       ↪ we don't need it
6  if !analyzePlatformInformation {
7      close(platformInformationChannel)
8  }
9
10 // go routine to analyze the repository using git independently from main
       ↪ thread
11 go func() {
12     routineContributionMap, routineErr := analyzeRepositoryFromString(
           ↪ repositoryUrl, commitsSince, commitsUntil, branch)
13     gitAnalyzationChannel <- GitAnalyzationChannel{
14         Result: routineContributionMap,
15         Reason: routineErr,
16     }
17     close(gitAnalyzationChannel)
18 }()
19
20 // execute go routine to fetch the platform information only when the
       ↪ platformInformation flag is set
21 if analyzePlatformInformation {
22     go func() {
23         routineIssues, routinePullRequests, routineErr :=
               ↪ getPlatformInformation(repositoryUrl, commitsSince,
               ↪ commitsUntil)
24         platformInformationChannel <- PlatformInformationChannel{
25             ResultIssues:       routineIssues,
26             ResultPullRequests: routinePullRequests,
27             Reason:             routineErr,
28         }
29         close(platformInformationChannel)
30     }()
31 }
32
33 // set the openness of the to the default value
34 chanel1Open := true
35 chanel2Open := analyzePlatformInformation
36
37 // initialize the return variables for the go routines
38 var contributionMap map[Contributor]Contribution
39 var issues []GQLIssue
40 var pullRequests []GQLPullRequest
41
42 // wait for the results of both go routines
43 for chanel1Open || chanel2Open {
44     select {
45     case msg1, ok1 := <-gitAnalyzationChannel:
46         if !ok1 {
47             // if the channel is closed set the flag to false
48             chanel1Open = false
49         } else if msg1.Reason != nil {
50             // error handling
51             if strings.Contains(msg1.Reason.Error(), "authentication") {
52                 makeHttpStatusErr(w, msg1.Reason.Error(), http.
                       ↪ StatusUnauthorized)
53             } else {
```

```
54                  makeHttpStatusErr(w, msg1.Reason.Error(), http.
                        ↪ StatusInternalServerError)
55              }
56              return
57          } else {
58              // save the return value to the initialized variable
59              contributionMap = msg1.Result
60          }
61      case msg2, ok2 := <-platformInformationChannel:
62          if !ok2 {
63              // if the channel is closed set the flag to false
64              chanel2Open = false
65          } else if msg2.Reason != nil {
66              // error handling
67              makeHttpStatusErr(w, msg2.Reason.Error(), http.
                    ↪ StatusInternalServerError)
68              return
69          } else {
70              // save the return value to the initialized variable
71              issues = msg2.ResultIssues
72              pullRequests = msg2.ResultPullRequests
73          }
74      }
75  }
```

Listing 4.11: Git Analysis and Platform Information concurrently

The performance analysis in Section 5.1.2 showed that analyzing the Git commits sequentially caused a considerable performance impact. To optimize this, the commits are analyzed concurrently with the help of go routines, and their results are sent back via channels. Go follows the principle that all channels should not be closed by a receiver because a go routine trying to send on a closed channel will cause an error. In the case of the analysis engine, however, this has to be done this way. Inside a go routine that analyzes the commit, it is impossible to say whether this go routine will be the last one sending on this channel. For this reason, the engine counts how many commits are analyzed (Line 2 & 4, Listing 4.13) and how many responses have already been sent over the channel (Line 68 & 89, Listing 4.13). When this number is equal, the channel is closed (Lines 90-92, Listing 4.13). Thereby it is ensured that in a go routine in each case exactly once is sent over a channel.

```
1  contributionChannel := make(chan ContributionChannel)
2  commitCounter := 0
3  err = commits.ForEach(func(c *object.Commit) error {
4      commitCounter++
5      go func() {
6          author := Contributor{
7              Name:  c.Author.Name,
8              Email: c.Author.Email,
9          }
10
11         merge := 0
12         commit := 1
13
14         if len(c.ParentHashes) > 1 {
```

```
15              merge = 1
16              commit = 0
17          }
18
19          stats, err := c.Stats()
20          if err != nil {
21              contributionChannel <- ContributionChannel{
22                  Result: Contribution{
23                      Contributor: Contributor{
24                          Name:  "",
25                          Email: "",
26                      },
27                      Changes:      CommitChange{
28                          Addition: 0,
29                          Deletion: 0,
30                      },
31                      Merges:       0,
32                      Commits:      0,
33                  },
34                  Reason: err,
35              }
36          } else {
37              changes := CommitChange{
38                  Addition: 0,
39                  Deletion: 0,
40              }
41
42              // count the lines if its not a merge, otherwise use a factor
43              if merge == 0 {
44                  for index := range stats {
45                      changes.Addition += stats[index].Addition
46                      changes.Deletion += stats[index].Deletion
47                  }
48              } else {
49                  for index := range stats {
50                      changes.Addition += int(float64(stats[index].Addition)
                            ↪ * mergedLinesWeight)
51                      changes.Deletion += int(float64(stats[index].Deletion)
                            ↪ * mergedLinesWeight)
52                  }
53              }
54
55              contributionChannel <- ContributionChannel{
56                  Result: Contribution{
57                      Contributor: author,
58                      Changes:     changes,
59                      Merges:      merge,
60                      Commits:     commit,
61                  },
62                  Reason: nil,
63              }
64          }
65      }()
66      return nil
67  })
68  answersReceived := 0
```

```
for res := range contributionChannel {
    if res.Reason != nil {
        return nil, err
    } else {
        author := res.Result.Contributor
        if _, found := authorMap[author]; !found {
            authorMap[author] = res.Result
        } else {
            authorMap[author] = Contribution{
                Contributor: author,
                Changes: CommitChange{
                    Addition: authorMap[author].Changes.Addition + res.
                        ↪ Result.Changes.Addition,
                    Deletion: authorMap[author].Changes.Deletion + res.
                        ↪ Result.Changes.Deletion,
                },
                Merges:  authorMap[author].Merges + res.Result.Merges,
                Commits: authorMap[author].Commits + res.Result.Commits,
            }
        }
    }

    answersReceived++
    if commitCounter == answersReceived {
        close(contributionChannel)
    }
}
```

Listing 4.12: Concurrency Adaption go-git

However, this performance optimization measure has a more vital stability impact on the developed software. The library go-git is not designed to perform this analysis concurrently. Go detects this weakness in the execution of tests. When performing the analysis with the API, Go does not detect this and allows the software to be executed. Only in very few cases (once in several million analyzed commits) this error could be reproduced. A modification of the go-git library, which foresees the use of a sync map instead of an ordinary map, has fixed this bug and enhanced the library (Listing 4.13).

```
"bytes"
    "io"
    "sort"
    "sync"

    encbin "encoding/binary"
@@ -56,7 +57,7 @@ type MemoryIndex struct {
    PackfileChecksum [20]byte
    IdxChecksum      [20]byte

    offsetHash       map[int64]plumbing.Hash
    offsetHash       sync.Map
    offsetHashIsFull bool
}
@@ -127,10 +128,7 @@ func (idx *MemoryIndex) FindOffset(h plumbing.Hash) (
    ↪ int64, error) {

```

```
17      if !idx.offsetHashIsFull {
18          // Save the offset for reverse lookup
19          if idx.offsetHash == nil {
20              idx.offsetHash = make(map[int64]plumbing.Hash)
21          }
22          idx.offsetHash[int64(offset)] = h
23          idx.offsetHash.Store(int64(offset), h)
24      }
25
26      return int64(offset), nil
27  @@ -169,39 +167,29 @@ func (idx *MemoryIndex) getCRC32(firstLevel,
        ↪ secondLevel int) uint32 {
28
29  // FindHash implements the Index interface.
30  func (idx *MemoryIndex) FindHash(o int64) (plumbing.Hash, error) {
31      var hash plumbing.Hash
32      var ok bool
33
34      if idx.offsetHash != nil {
35          if hash, ok = idx.offsetHash[o]; ok {
36              return hash, nil
37          }
38      if hash, ok := idx.offsetHash.Load(o); ok {
39          return hash.(plumbing.Hash), nil
40      }
41
42      // Lazily generate the reverse offset/hash map if required.
43      if !idx.offsetHashIsFull || idx.offsetHash == nil {
44      if !idx.offsetHashIsFull {
45          if err := idx.genOffsetHash(); err != nil {
46              return plumbing.ZeroHash, err
47          }
48
49          hash, ok = idx.offsetHash[o]
50      }
51
52      if !ok {
53          return plumbing.ZeroHash, plumbing.ErrObjectNotFound
54          if hash, ok := idx.offsetHash.Load(o); !ok {
55              return plumbing.ZeroHash, plumbing.ErrObjectNotFound
56          } else {
57              return hash.(plumbing.Hash), nil
58          }
59      }
60
61      return hash, nil
62      return plumbing.Hash{}, nil
63  }
64
65  // genOffsetHash generates the offset/hash mapping for reverse search.
66  func (idx *MemoryIndex) genOffsetHash() error {
67      count, err := idx.Count()
68      if err != nil {
69          return err
70      }
71
```

```
72      idx.offsetHash = make(map[int64]plumbing.Hash, count)
73      idx.offsetHashIsFull = true
74
75      var hash plumbing.Hash
76  @@ -211,7 +199,7 @@ func (idx *MemoryIndex) genOffsetHash() error {
77          for secondLevel := uint32(0); i < fanoutValue; i++ {
78              copy(hash[:], idx.Names[mappedFirstLevel][secondLevel*
                    ↪ objectIDLength:])
79              offset := int64(idx.getOffset(mappedFirstLevel, int(secondLevel
                    ↪ )))
80              idx.offsetHash[offset] = hash
81              idx.offsetHash.Store(offset, hash)
82              secondLevel++
83          }
84      }
```

Listing 4.13: Concurrency Adaption go-git Library

4.3.5 Requests

The analysis engine has two endpoints from which the data can be retrieved. The first one collects only the contributions information needed for the evaluation and returns this list. Depending on whether the platform information is to be included in the analysis, it will also be included in the response.

GET /contributions

Collected contribution metrics for each Git user

Parameters

repositoryUrl	*required*	url of .git file of the repository
since	*optional*	date at which the analysis should start in RFC3339 format
until	*optinoal*	date at which the analysis should end in RFC3339 format
platformInformation	*optinoal*	flag whether platform information such as issues and pull requests should be analyzed
branch	*optinoal*	name of the branch that should be analyzed

Response

```
1  [ // list of each contributing git user
2    {
3      "gitInformation": {
4        "contributor": {
5          "name": string // name of the git user,
6          "email": string // email of the git user
7        },
```

```
8         "changes": {
9           "addition": int // added lines,
10          "deletion": int // deleted lines
11        },
12        "merges": int // amount of merges,
13        "commits": int // amount of commits
14      },
15      "platformInformation": {
16        "userName": string // platform username,
17        "issueInformation": {
18          "author": []int // list of amount of comments of each created issue
19          "commenter": int // amount of comments written
20        },
21        "pullRequestInformation": {
22          "author": [ // list of pull requests the author created
23            {
24              "state": string // current state of pull request
25              "reviews": []string // reviewing events on pull request
26            }
27          ]
28          "reviewer": int // amount of reviews written
29        }
30      }
31    }
32  ]
```

Listing 4.14: Response Structure Collected Metrics

The second endpoint returns the results as a share of the total contribution for each Git contributor.

GET	/weights

Listing of the share of the total contribution by each Git user

Parameters

repositoryUrl	*required*	url of .git file of the repository
since	*optional*	date at which the analysis should start in RFC3339 format
until	*optinoal*	date at which the analysis should end in RFC3339 format
platformInformation	*optinoal*	flag whether platform information such as issues and pull requests should be analyzed
branch	*optinoal*	name of the branch that should be analyzed

Response

```
1  [ // list of each contributing git user
2    {
3      "contributor": {
4        "name": string // name of the git user,
5        "email": string // email of the git user
```

```
6       },
7       "weight": float // [0,1] representing the share of the contribution
8     }
9 ]
```

Listing 4.15: Response Structure Computed Score

Chapter 5

Evaluation and Discussion

This chapter is dedicated to the evaluation of the developed analysis engine. In the first part, different areas of the program are examined for their performance. The goal was to find out which executions influence the performance most and if they can be optimized.

The second part describes the process of the certification of the weigths, which is needed for the developed Equation 4.1 to calculate the contribution. On a use-case the analysis engine was tested with a repository. The results were discussed with the corresponding repository owners. In an iterative process the weights were adjusted so that the calculated contribution matched the perceived contribution of the repository owners.

5.1 Performance Testing

To test the performance of the analysis engine, different aspects of the engine were examined. Specifically, the following aspects:

- Cloning and updating the repository
- Analyzing the repository in a way to determine the performance impact of each metrics
  - Once we will analyze the whole repository
  - Once we will analyze a real-world usage example of analyzing a time period of 3 months
- Fetching platform data (*e.g.,* GitHub)
- Mapping platform to the information collected from Git
- Calculating the contribution score with the formula that weights the raw contribution of each developer

These performance tests were executed on real-world examples of open source repositories. Four projects were selected to represent different groups of repository size, small, medium, large, very large. The smallest group of repositories will be represented by the moment.js [33] project on the moment repository from moment on GitHub. This is a JavaScript library to format data, which is very popular in the JavaScript developer community. At the time of measurement, there were 3'953 commits in the history of the master branch.

The react native project on Facebook's GitHub repository react-native [7] will represent the next size of the repositories. This project, which is still in development for years, is based on the popular JavaScript library react and uses this structure to develop native apps for mobile devices. At the time of measurement, there were 21'167 commits in the history of the master branch.

Nextcloud is one of the larger open source projects of today [34]. As a spin-off and further development of the ownCloud [37] project, Nextcloud wants to make the cloud universe open-source and self-hostable. In addition to the server repositories from Nextcloud to GitHub that are being investigated in this context, further open-source projects are being developed to complement the server's functionality. At the time of the measurement, there were 55'546 commits in the history of the master branch.

The largest repository examined in this measurement is the rust programming language [41]. With 129'179 commits on the master branch at the time of the measurement, this repository represents the largest open-source repositories. The Linux repository is not examined as a representation of huge repositories. Although it is much larger than the rust repository with 980'000 commits, there are hardly any other open-source repositories of this size, which could be represented by the Linux repository.

All the mentioned performance tests were executed locally on the same computer. Goland [26] was executed on the computer in whose console the analysis engine was running. Besides, the postman [40] program was executed, which sends the requests to the REST API. In order to record the results, Google Sheets was executed in a browser. Go internal timing tools determined the duration of the execution of each aspect in milliseconds (ms). The specifications of the hardware utilized during the tests were an Intel (R) Core (TM) i7-4771 @ 3.50 GHz with 32 GB of RAM, and Git version `2.17.1.windows.2`.

5.1.1 Cloning / Updating Repository

Cloning

The first test evaluated the implemented analysis engine's performance when cloning a repository that is not yet locally present (*i.e.,* stored in the local machine). The engines cloned the repository with the configuration of a single branch repository and its selected branch. Each of the repositories listed in the last section was cloned ten times to estimate the upper and lower bounds for the execution time. As expected, the cloning duration depends on the repository size, as depicted in Figure 5.1. There, the x-axis represents the different repositories, and the y-axis represents the duration of the repository cloning to the local disk. It has also been noticed that multiple consecutive cloning of large

repositories became a longitudinal process. For this reason, the repositories were cloned over a more extended period of time to avoid throttling. This should also provide a fair comparison between the cloning durations.

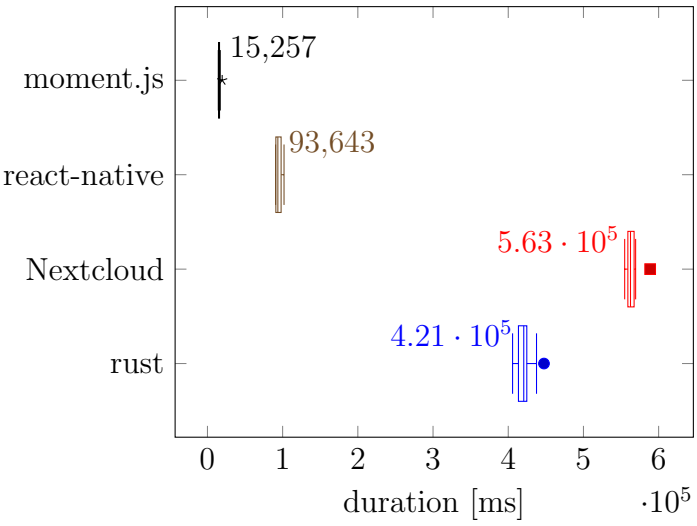

Figure 5.1: Execution Time: Cloning Repository

Updates

If the repository already exists locally, it will only search for updates. It is possible that in such a case, new commits will be fetched because the local branch is no longer up to date with the remote branch or because a new branch needs to be fetched because the branch under investigation has switched since cloning. In the performance analysis, only the case that no new commits are available was measured, *i.e.,* the pure checking of updates. In the event that new commits were available, they could not be reliably tested multiple times so that each test would be performed under the same conditions.

To measure this process, the measurement was divided into three different test executions to draw better conclusions. In the first test execution of this process, the duration of the repository's pure opening was tested. The tests show an execution time of 0 to 1 ms regardless of the size of the repository. In the second test execution, the checking of updates was added after opening the repository (Figure 5.2). Finally, in the third test execution, it was checked how long it takes to open the repository and select the correct branch (Figure 5.3). It turned out that most of the time was due to selecting the correct branch. Opening the repositories and checking for updates induced only a small part of the total duration.

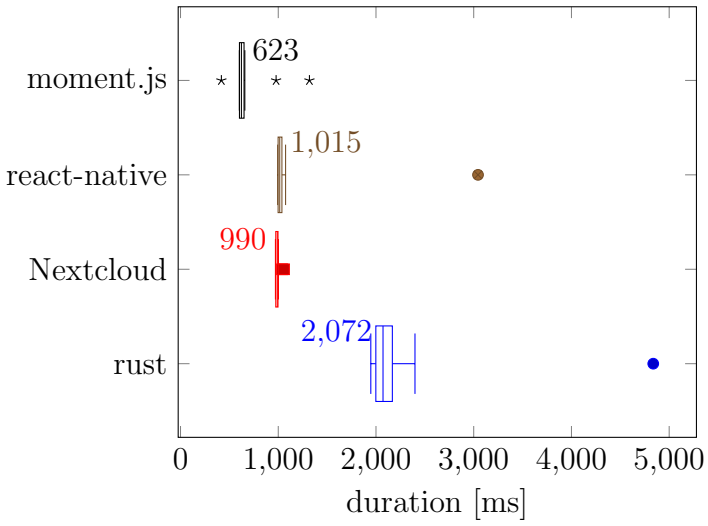

Figure 5.2: Execution Time: Checking for Updated Repository Version

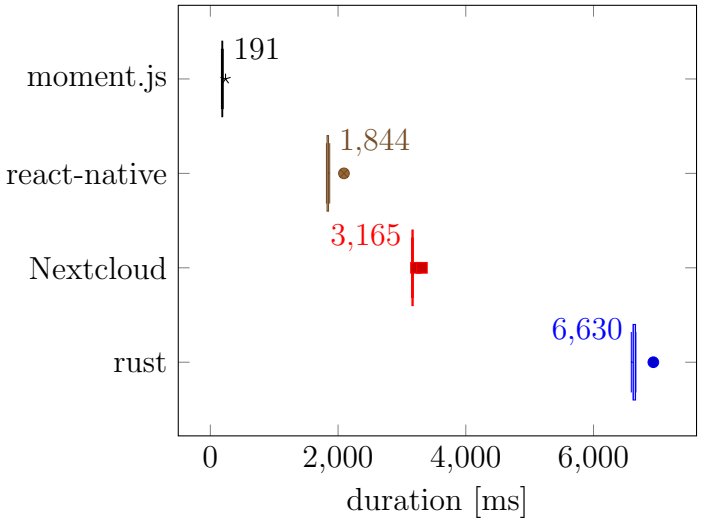

Figure 5.3: Execution Time: Checking Out Correct Branch of Repository

5.1.2 Analyzing the Repository

All Metrics

When the first tests of all metrics were carried out for a time period of three months, it was discovered that an analysis of 690 commit already took about 80 seconds. With a linear projection to repositories of 120'000 commits, this would correspond to a duration of about 4.5 hours. A further check of the code for optimization possibilities showed that a parallel extraction of the metrics from a commit could reduce this time massively. An implementation with go routines, which allow this concurrency, allowed to reduce the duration from about 80 seconds to 25 seconds. All other tests were executed with the implementation of go routines.

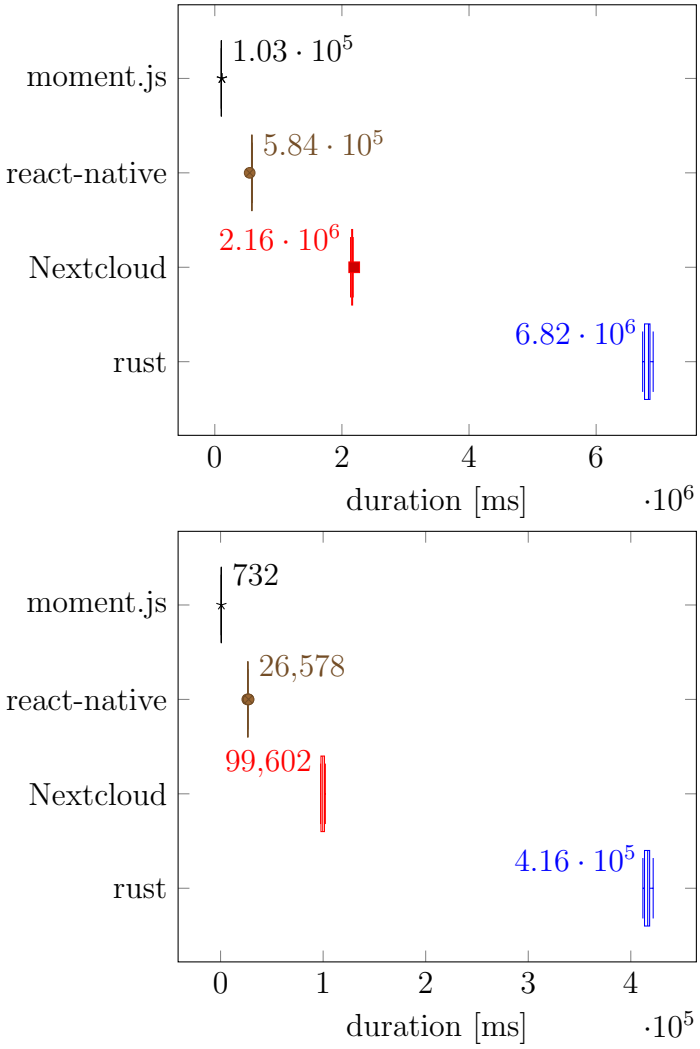

Figure 5.4: Execution Time: Collecting All Git Metrics (Lifetime vs. 3 Months)

No Metrics

This test showed the amount of time an analysis takes without extracting any metrics from the commits. The engine was only looping through the commits. The analysis showed a dependence on the number of commits. This correlation was most significant in the analysis of the total lifespan of the repository. An average duration per commit of [0.12, 0.13] ms was calculated for all four analyzed repositories. However, this ratio was not as constant when calculating a three-month period of the repository. Moment.js had an average of 14.64 ms per commit, while react-native had one of 3.85 ms, nextcloud one of 4.87 ms, and rust one of 2.62 ms. Therefore, the analysis engine only develops its performance after a certain number of commits, and that the limitation to a specific time period also influences the performance.

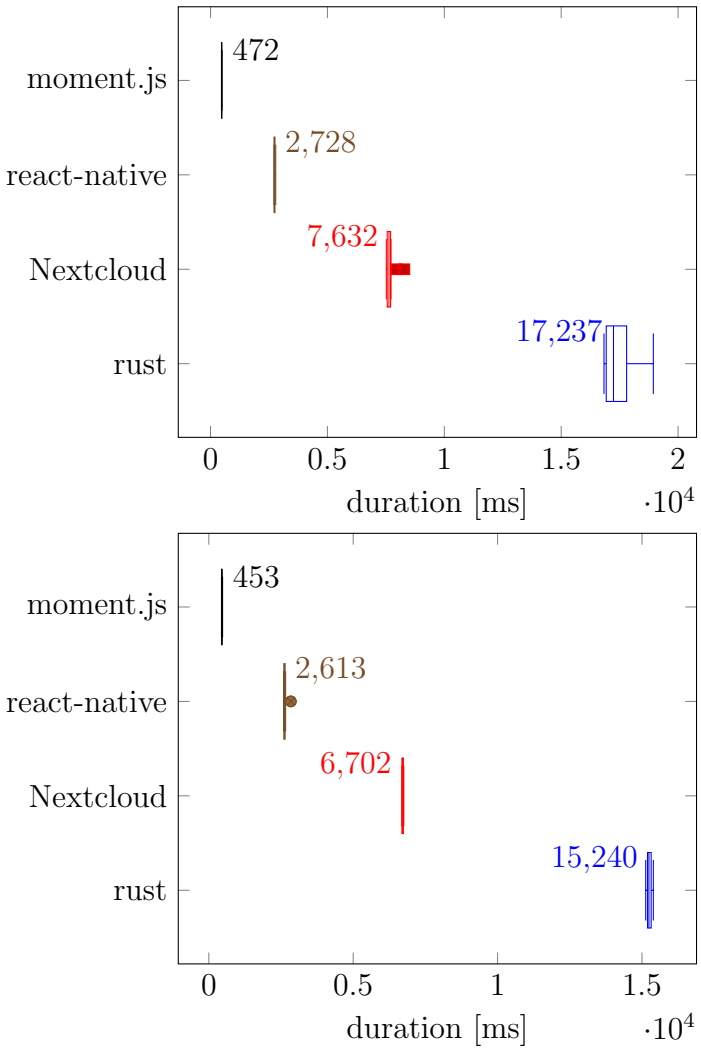

Figure 5.5: Execution Time: Collecting No Git Metrics (Lifetime vs. 3 Months)

Only Git History (commits/merges)

In addition to the previous test execution with no metrics, this one distinguishes between commits and merges. Figure 5.6 clearly shows that this distinction has no significant performance impact on the analysis engine.

Only Changes (additions/deletions)

During the execution of the test, where the effect of the examination of changed lines is evaluated, in addition to the normal looping through of the commits, the changed commits are added up for each one. This query caused the most extraordinary performance impact in the analysis (Figure 5.7). While the analysis of small repositories only takes a little more than a minute over their entire lifetime, an analysis of a huge repository takes more than 1.5 hours. An analysis of small and medium-sized repositories over a period of 3 months can significantly reduce this execution time so that only seconds are required. For

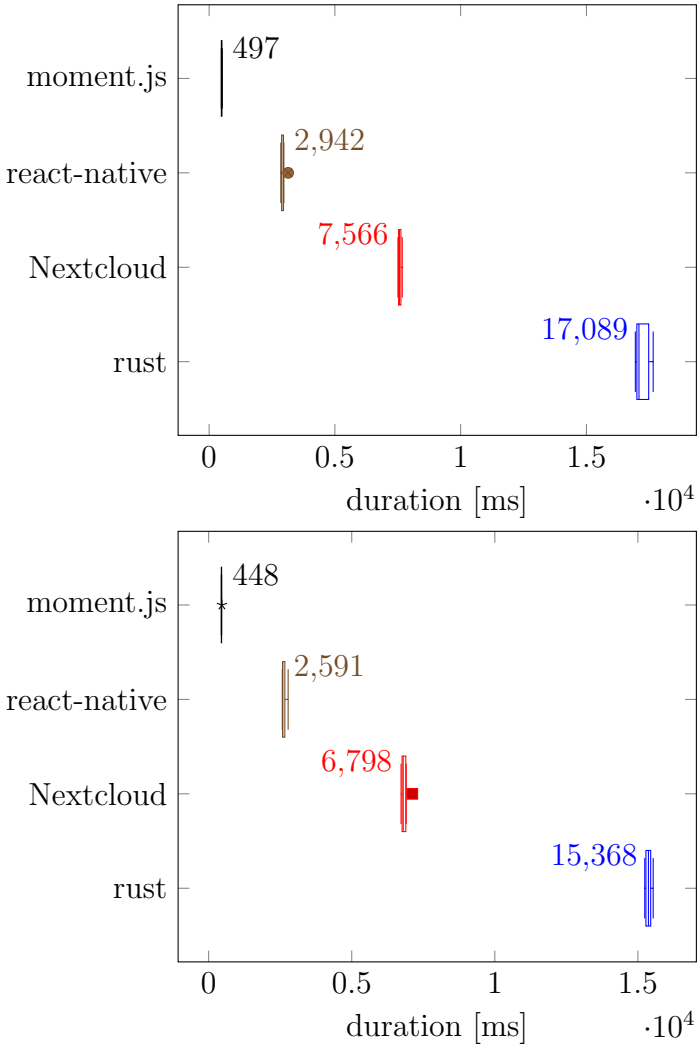

Figure 5.6: Execution Time: Collecting History as Git Metrics (Lifetime vs. 3 Months)

large and very large repositories with much activity, an analysis over a shorter period of time must also be expected to take several minutes.

5.1.3 Platform Information

Due to the rate limits of the platform information retrieval from GitHub, only a period of 3 months was considered in this analysis. A large repository like the one chosen by rust now has over 36'000 issues and 41'000 pull requests. A fetch of all this data and allocate it to Git users drives the number of requests to the limit. An analysis of such a repository over the whole runtime can, therefore, not be done with the chosen data retrieval method or if the contingent of available requests is just sufficient, at least only sporadically.

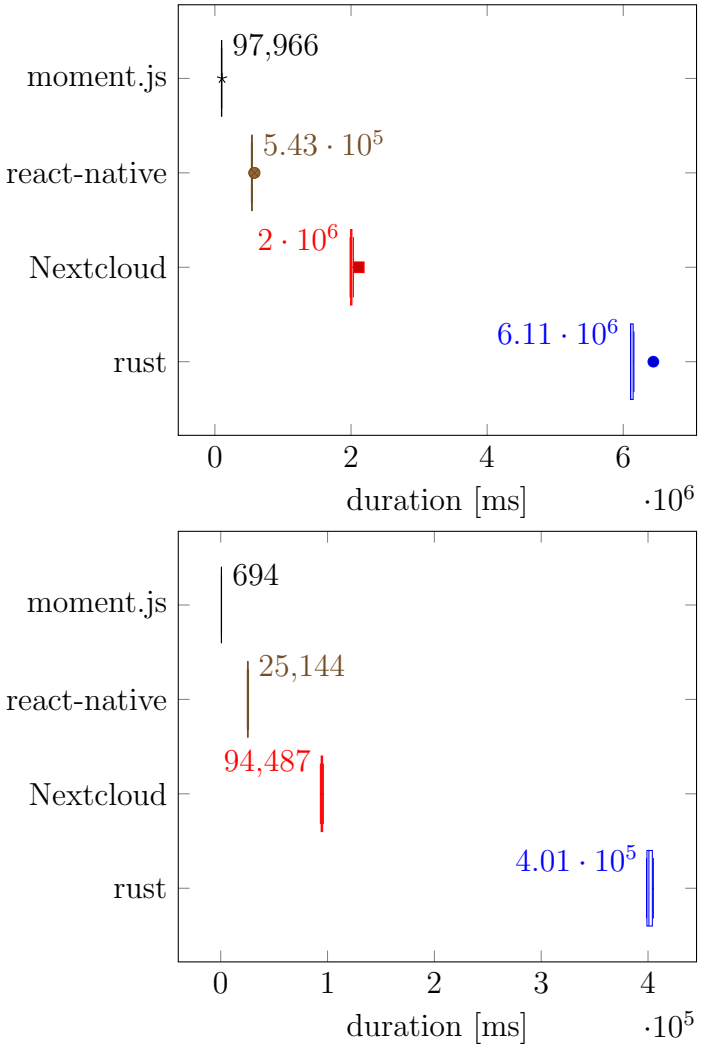

Figure 5.7: Execution Time: Collecting Changed Lines as Git Metrics (Lifetime vs. 3 Months)

Collecting Information

The results of the analysis of the duration of the collection of platform information from GitHub show a result correlated with the number of issues and pull requests. However, there were differences in the breakdown of the average time to obtain a commit/pull request. This duration ranges between 82 ms and 92 ms, whereas the rust repository with the highest number of issues and pull requests has a higher duration of 116 ms. The reason for this is the number of requests. Issues, issue comments as well as pull requests and their activities can be requested from GitHub in a maximum bundle of 100. To get more data, a pagination approach is used to get the missing data from the API. If more requests are made, because the page length is often only just exceeded, the duration per issue/pull request increases.

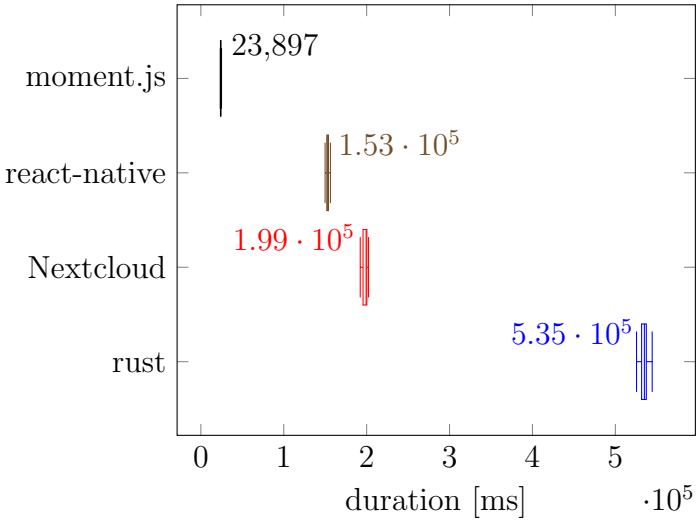

Figure 5.8: Execution Time: Collecting Platform Information (3 Months)

Mapping to Git Users

For mapping the collected platform information to a Git user, requests to GitHub are also necessary. To be precise, one request per Git user is needed. The tests confirmed the assumption that the time needed for mapping increases with the number of contributors. Nevertheless, it is interesting to note that the mapping duration is not constant when calculated down to one Git user. With a duration of 278 ms to 577 ms per Git user, it is so different that no reliable prediction can be made for a different number of users.

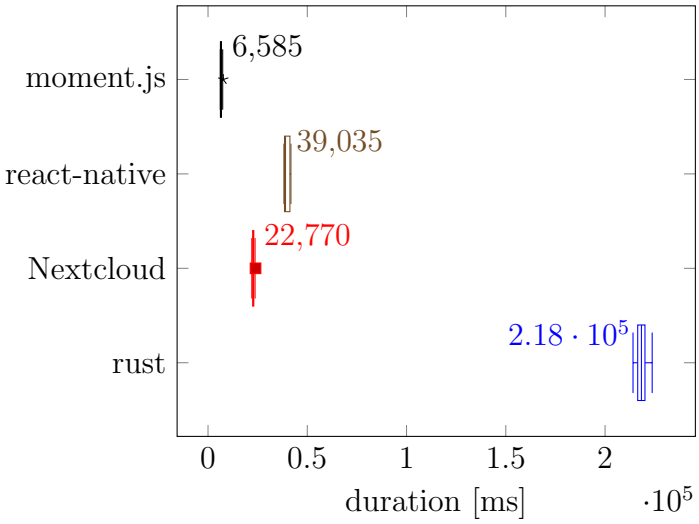

Figure 5.9: Execution Time: Mapping Platform Information to Git Users (3 Months)

5.1.4 Weight Analysis

Without Platform Information

The weight analysis only involves applying the defined formula to the collected information. This calculation took only between 0 ms and 3 ms and is therefore negligible.

With Platform Information

The weight analysis with platform information takes measurably longer than without platform information. For large repositories, the runtime of this process increases up to 20 ms. However, it should be noted that this noticeable increase in runtime was only measurable in an analysis over the entire lifetime of the repository. In an analysis over a shorter time frame such as three months, no noticeable effects were found.

5.2 Use Case

To present a use case of the analysis engine and evaluate the results, it has to be displayed on a repository with an adequate number of developers involved. A high number of contributors would unnecessarily extend the length of the analysis engine's response in this example. Furthermore, assessing the correctness of the evaluated contribution by the persons involved in the repository would be more difficult and confusing with more contributors. To assess the results of the engine, the persons involved in the repository are expected to be able to evaluate the work of all contributors. Since personal acquaintances to the main contributors of the neow3j repository were available, this was chosen as an application and evaluation example.

For the analysis of the repository, the following request was sent to the analysis engine. With this request, the neow3j repository on the master-3.x branch (the current master branch) should be analyzed from January 1st to July 1st. Furthermore, an overview of the contribution should be given.

```
GET http://localhost:8080/weights?
          repositoryUrl=https://github.com/neow3j/neow3j.git
          &since=2020-01-01T00:00:00Z
          &until=2020-07-01T00:00:00Z
          &platformInformation=true
          &branch=master-3.x
```

Figure 5.10: Request to Analyze neow3j Repository

```
[
    {
        "contributor": {
            "name": "Guilherme Sperb Machado",
            "email": "guil@axlabs.com"
        },
        "weight": 0.29578597951016683
    },
    {
        "contributor": {
            "name": "Claude Muller",
            "email": "claude@axlabs.com"
        },
        "weight": 0.5499743238994934
    },
    {
        "contributor": {
            "name": "Guil. Sperb Machado",
            "email": "guil@axlabs.com"
        },
        "weight": 0.016488273578008774
    },
    {
        "contributor": {
            "name": "mialbu",
            "email": "michael@axlabs.com"
        },
        "weight": 0.011311198591179652
    },
    {
        "contributor": {
            "name": "Michael Bucher",
            "email": "michael@axlabs.com"
        },
        "weight": 0.12644022442115138
    }
]
```


Listing 5.1: Response of Analyzation With Platform Information


```
[
    {
        "contributor": {
            "name": "Michael Bucher",
            "email": "michael@axlabs.com"
        },
        "weight": 0.19191669083734353
    },
    {
        "contributor": {
            "name": "Claude Muller",
            "email": "claude@axlabs.com"
        },
        "weight": 0.6048589289082649
    },
    {
```

```
        "contributor": {
            "name": "Guilherme Sperb Machado",
            "email": "guil@axlabs.com"
        },
        "weight": 0.1610851634356544
    },
    {
        "contributor": {
            "name": "Guil. Sperb Machado",
            "email": "guil@axlabs.com"
        },
        "weight": 0.025056728126830102
    },
    {
        "contributor": {
            "name": "mialbu",
            "email": "michael@axlabs.com"
        },
        "weight": 0.017082488691907174
    }
]
```


Listing 5.2: Response of Analyzation Without Platform Information

These responses indicate which Git contributor was responsible for which partition of the total contribution in the selected time period. As seen in the outcome in the response, there are some very similar sounding users. This is because a developer can have several Git accounts. If a developer makes changes directly on a platform like GitHub, this may cause a different Git user to be created, even though it is the same developer. Therefore, in the following charts, the Git users belonging to one developer are combined and their shares are summed up.

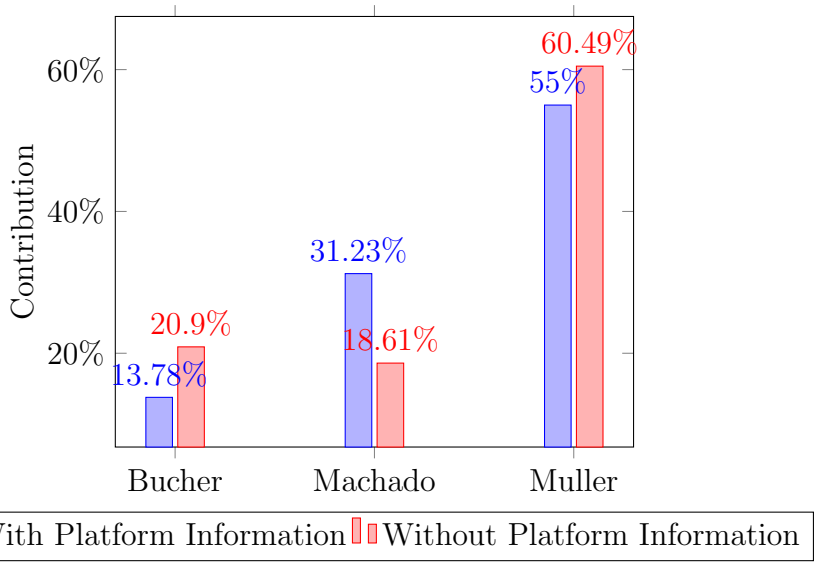

Figure 5.11: Share of Contribution neow3j

This summary of the contribution was conducted within multiple iterations where in each iteration the results were discussed with the repository owner. Consequently, the

weights were adapted. The key insight from the feedback and adaptation was that the platform information was not sufficiently weighted. In the end the analysis was regarded as representative of the actual contribution. Figure 5.11 shows the contribution per developer in the first semester of 2020, where once platform information was considered and the other time not.

5.3 Discussion

The analysis engine developed in this study gives a new possibility to determine the contribution in open source projects. With the chosen metrics, the available and reasonably assessable ones were chosen to determine the contribution. There are undoubtedly other metrics that could determine the contribution more accurately, but the limitations of open source repositories and data availability limit their inclusion. In earlier literature, other metrics can also be found that can be applied to open source repositories. However, in the literature, these are only applied to a project in one language. Besides, often things have to be configured for these analyses. Therefore the selection of metrics seems to be reasonable.

Performance-wise, the metrics for LOC and platform information stand out. However, these are precisely the central metrics for estimating the contribution. In earlier literature, the LOCs are used most of the time. In the use-case analysis with the discussion with the owners of the neow3j repository, the platform information should not be omitted either, since there is also valuable information available for the work done.

With quality metrics that can be automatically assessed and applied to all repositories, regardless of language and technology, and attributed to individual contributors, there would be an even better overview of the contribution. Furthermore, with the defined weights, which were refined in the use-case, they cannot be assumed to apply equally to every project. Possibly the weights would have been defined slightly differently if the use-case had been executed with other repositories. However, the defined weights are certainly a good starting point. Finally, the analysis engine is still an engine that tries to estimate the contribution with the available data.

Chapter 6

Summary and Conclusions

In this thesis, a contributions analysis engine for open source software projects was presented. It was shown which existing approaches for contribution analysis exist and which metrics were used.

Furthermore, different metrics for this engine were discussed in detail, and the final selection of the metrics used was explained. Three criteria mainly determined this selection of metrics. The first significant limitation was the availability of information. All open-source projects should be able to be analyzed, so the data source is limited to publicly available information. The second major limitation was that it is possible to analyze any open source project. Metrics that are only applicable to individual projects are, therefore, not taken into account. The third was the automatic analysis. It should be possible to analyze a project with simplicity and without the need for configuration. With these limitations, the following metrics were selected. On top of metrics extracted from Git, namely commits, merges, additions, and deletions, the metrics issues and pull requests from platforms are optionally included in the evaluation.

Thereupon, a suitable formula was developed that uses the metrics selected from these to make a statement about the developers' contribution. This formula was validated with open source contributors on a use-case. A performance evaluation identifies areas of the engine that have a severe impact on the runtime. Based on this evaluation, an optimization of the code was possible.

Future Work

The developed engine brings to the previous methods a possibility to analyze open source repositories. The disadvantage of this engine is the calculation based on simple metrics. For future work, it is planned be helpful to research the quality of a contribution. The development of such a tool, independent of language and project, could make a statement about the quality of a contribution without prior configuration. Moreover, a suitable solution for requests must be found before integrating the analysis engine into an existing system. A REST API interface offers some disadvantages for requests that take up to

several minutes to respond. A scheduler could also help to analyze several open source projects on schedule.

Bibliography

[1] Atlassian. Bitbucket, 2020. https://bitbucket.org Last visit October 16, 2020.

[2] Victor R. Basili, Richard W. Selby, and T Phillips. Metric analysis and data validation across fortran projects. *IEEE Transactions on Software Engineering*, (6):652–663, 1983.

[3] X. Ben, S. Beijun, and Y. Weicheng. Mining Developer Contribution in Open Source Software Using Visualization Techniques. In *Third International Conference on Intelligent System Design and Engineering Applications (ICISDEA 2013)*, pages 934–937, 2013.

[4] C. Berge and E. Minieka. *Graphs and Hypergraphs*. Graphs and Hypergraphs. North-Holland Publishing Company, 1973.

[5] Jailton Coelho and Marco Tulio Valente. Why Modern Open Source Projects Fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 186–196, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Nadia Eghbal. Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure. Technical Report. Ford Foundation, 2016. https://www.fordfoundation.org/media/2976/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure.pdf Last visit September 30th, 2020.

[7] Facebook Inc. React GitHub Repository, 2020. https://github.com/facebook/react Last visit September 30th, 2020.

[8] International Organization for Standardization and International Electrotechnical Commission. *Software Engineering-Product Quality*, volume 9126. ISO/IEC, 2001.

[9] Git. Embedding Git in Your Applications - go-git. https://git-scm.com/book/en/v2/Appendix-B:-Embedding-Git-in-your-Applications-go-git Last visit September 22, 2020.

[10] Git. Git - About Version Control, 2020. https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control Last visit October 5, 2020.

[11] Git. Git - Branches in a Nutshell, 2020. https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell Last visit October 14, 2020.

[12] Gitea. Gitea - A Painless Self-Hosted Git Service, 2020. https://gitea.io/en-us/ Last visit October 11, 2020.

[13] GitHub. Github help - creating an issue, 2020. https://docs.github.com/en/enterprise/2.15/user/articles/creating-an-issue Last visit August 23, 2020.

[14] GitHub. Github search - users, 2020. https://github.com/search?q=type:user&type=Users Last visit August 8, 2020.

[15] GitHub. GitHub Sponsors, 2020. https://github.com/sponsors Last visit October 20, 2020.

[16] GitHub Inc. GitHub, 2020. https://github.com Last visit October 16, 2020.

[17] GitLab. GitLab, 2020. https://gitlab.com Last visit October 16, 2020.

[18] go git. go-git/go-git. https://github.com/go-git/go-git Last visit September 22, 2020.

[19] golang. GOPATH - Golang/Go Wiki, 2020. https://github.com/golang/go/wiki/GOPATH Last visit October 5, 2020.

[20] gorilla. gorilla/mux. https://github.com/gorilla/mux Last visit September 22, 2020.

[21] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring Developer Contribution from Software Repository Data. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 129–132, New York, NY, USA, 2008. Association for Computing Machinery.

[22] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring Developer Contribution from Software Repository Data. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, 2008.

[23] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[24] Péter Hegedüs, T Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. A Drill-Down Approach for Measuring Maintainability at Source Code Element Level. 01 2013.

[25] ISO. Iec25010: 2011 Systems and Software Engineering–Systems and Software Quality Requirements and Evaluation (SQUARE)–System and Software Quality Models. *International Organization for Standardization*, 34:2910, 2011.

[26] JetBrains. GoLand: A Clever IDE to Go by JetBrains, 2020. https://www.jetbrains.com/go/ Last visit October 11, 2020.

[27] Khari Johnson. GitHub Passes 100 Million Repositories, 2018. https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/ Last visit October 16, 2020.

[28] Stephen H Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2003.

[29] Zhifang Liao, Benhong Zhao, Shengzong Liu, Haozhi Jin, Dayu He, Liu Yang, Yan Zhang, and Jinsong Wu. A Prediction Model of the Project Life-Span in Open Source Software Ecosystem. *Mob. Netw. Appl.*, 24(4):1382â1391, August 2019.

[30] J. Lima, C. Treude, F. F. Filho, and U. Kulesza. Assessing developer contribution with repository mining-based metrics. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 536–540, 2015.

[31] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[32] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[33] Moment.js. Moment.js GitHub Repository, 2020. `https://github.com/moment/moment` Last visit September 30th, 2020.

[34] Nextcloud GmbH. Nextcloud GitHub Repository, 2020. `https://github.com/nextcloud` Last visit September 30th, 2020.

[35] Open Collective. Open Collective - Make your community sustainable. Collect and spend money transparently., 2020. `https://opencollective.com` Last visit October 20, 2020.

[36] Enrique Ivan Oviedo. Control flow, data flow and program complexity. 1984.

[37] ownCloud GmbH. ownCloud - share files and folders, easy and secure, 2020. `https://owncloud.com/` Last visit October 27th, 2020.

[38] R. M. Parizi, P. Spoletini, and A. Singh. Measuring Team Members' Contributions in Software Engineering Projects using Git-driven Technology. In *IEEE Frontiers in Education Conference (FIE 2018)*, pages 1–5, San Jose, CA, USA, USA, October 2018.

[39] Patreon. Patreon, 2020. `https://patreon.com` Last visit October 20, 2020.

[40] Postman. Postman | The Collaboration Platform for API Development, 2020. `https://www.postman.com/` Last visit October 11, 2020.

[41] Rust Team. Rust Programming Language GitHub Repository, 2020. `https://github.com/rust-lang/rust` Last visit September 30th, 2020.

[42] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. Code Quality Analysis in Open Source Software Development. *Information systems journal*, 12(1):43–60, 2002.

[43] Linus Torvalds. Tech Talk: Linus Torvalds on git. `https://www.youtube.com/watch?v=4XpnKHJAok8` Last visit October 16, 2020.

[44] Elaine J Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

List of Figures

List of Tables

Appendix A

Installation Guidelines

A.1 Server Installation

If you are a windows user, you can simply execute the file engine.exe. This will allow the server to run locally and handle requests to the endpoints.

Otherwise you need to synchronize the dependencies first. As this project is set up with a go.mod file, this can be done by executing the following command inside the project directory.

```
go install
```

Afterwards, you can build the engine by executing the following command inside the project directory.

```
go build
```

This results in a binary, that can be executed to run the server.

A.2 Usage

The project is configured that way, that it runs the server at port 8080 of localhost. To call an endpoint simply make a GET request to the server at localhost:8080 with one of the following endpoints.

/contributions Request a listing of all contributors with their contributions

/weights Request a listing of the share of the total contribution by each contributor

parameters

- repositoryUrl

  - required
  - link to .git file or the repository
  - example: repositoryUrl=https://github.com/neow3j/neow3j.git

- since
  - optional (default: date of first commit)
  - date at which the analysis should start in RFC3339 format.
  - example: since=2020-01-22T15:04:05Z

- until
  - optional (default: now)
  - date at which the analysis should end in RFC3339 format.
  - example: until=2020-07-12T11:34:55Z

- platformInformation
  - optional (default: false)
  - flag whether platform information such as issues and pull requests should be analyzed.
  - example: platformInformation=true

- branch
  - optional (default: master)
  - name of the branch that should be analyzed.
  - example: branch=develop

Example

```
GET http://localhost:8080/weights?
            repositoryUrl=https://github.com/neow3j/neow3j.git
            &since=2020-01-01T00:00:00Z
            &until=2020-07-01T00:00:00Z
            &platformInformation=true
            &branch=master-3.x
```

Appendix B

Contents of the CD

- ZIP-file containing the project repository
- executable file of the engine for the windows platform
- PDF of the thesis