Creating, Measuring and Evaluating a Metric for Software Engineering Performance

MASTER THESIS

Thomas Wolter

Submitted on 20 December 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Stefan Buchner
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 20 December 2022

License

Erlangen, 20 December 2022

Abstract

Software engineering productivity is a metric that can be an indicator for software quality, development efficiency and developer satisfaction. As a result, the metric can be used for managerial decision making and is crucial to the success of a software project. However, even though the importance of the topic is widely accepted, there is no uniform approach to measuring it. In this thesis, we use a design science approach to provide a new objective metric to measuring software engineering productivity based on past literature. The approach uses openly available repository metadata collected by Git and GitHub and is usable for both a classic software engineering approach and an inner source approach. Furthermore, we demonstrate the functionality of our approach by analyzing a software engineering project.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

SLOC Source Lines of Code

CSV Comma-separated values

1 Introduction

Software engineering productivity can be used as an indicator for software quality, development efficiency and developer satisfaction (Forsgren et al., 2021; Graziotin & Fagerholm, 2019). Thus, being able to measure software engineering productivity in one's company or software project can be crucial to managerial decision making and the overall success of software engineering projects (Oliveira et al., 2016). However, even though the importance of measuring productivity is clear, no consensus on how to achieve this has been made. In fact, researchers have called for finding a more uniform approach and reviewed the extensive body of literature (Chapetta & Travassos, 2016; Forsgren et al., 2021; Wagner & Ruhe, 2018).

Existing approaches most commonly diverge in the the data sources used during the evaluation. Many early attempts were for example based on the added Source Lines of Code (SLOC) in a specific time frame. This however neglects the wealth of data available and the complexity of the issue (Murphy-Hill et al., 2019). Modern research for example attempt to use more complex indicators such as bugfixing work and developer surveys (Diamantopoulos et al., 2020; Meyer et al., 2014).

One factor that has become increasingly relevant is data gathered through repository mining. A developer committing software code to a repository creates metadata with information on additions. Furthermore, code hosting platforms, such as GitHub, also generate metadata when discussions take place. This metadata can be analyzed by parsing the data and provides an openly available source for measuring productivity. For this thesis we focus on creating an approach using metadata generated during the software engineering process.

This thesis contributes to the topic, by analyzing the existing body of work and generating an approach to measuring software engineering productivity based on the findings. Furthermore, we demonstrate this approach, by analyzing a software engineering project consisting of 8 repositories. The approach is usable for both a dedicated and an inner source software engineering approach and can be used to enable a comparison between the two.

For a research approach we chose to follow the design science methodology. This choice was made, because measuring software engineering productivity is a mostly unsolved identified organizational problem, which design science is focused on solving (Hevner et al., 2004). In detail, we follow the approach defined by Peffers et al. (2007) and structure the thesis as follows:

- Firstly, in chapter 2 we give an overview of related literature and define the problem we intend to solve.
- In chapter 3 we define the desired requirements we expect our solution to fulfill.
- In chapter 4 we describe our approach to the design and development of our solution.
- In chapter 5 we describe the tools we used and the steps we took to implement our solution.
- In chapter 6 we demonstrate our solution by analyzing a sample of open source projects.
- In chapter 7 we compare the desired results of our artifact with the actual outcome from chapter 6.
- Lastly, in chapter 8 we give our concluding thoughts on the thesis.

2 Problem Identification

As described in the previous section, measuring productivity in software engineering is not an easy task. There are many factors and challenges influencing a software developers work. This section will define key terminology, describe potential problems in detail and give background information by analyzing previous literature.

2.1 Productivity

2.1.1 Definition

The term *productivity* has been used in a variety of different contexts in software engineering research. In our thesis we rule out the use of productivity for topics such as determining a software tool's contribution to the overall workflow or the reuse of past software projects. Instead, we focus on a developer's contribution to a software project. *Thus, we define software engineering productivity as the efficiency of the output resulting from software developers contributions.* Furthermore, it is important to mention that this thesis is not focused on investigating ways to enhance productivity in software engineering. Our purpose is finding ways to measure productivity in software engineering projects.

2.1.2 Problems Measuring Productivity

Software developer productivity is of great importance to software engineering, because it can give information about a software project's status throughout the entire development cycle. This can for example be used to determine development efficiency, developer satisfaction and influence managerial decision making (Forsgren et al., 2021; Graziotin & Fagerholm, 2019; Oliveira et al., 2016). As such, it is important to find ways to measure productivity. However, there are many challenges to measuring productivity.

Firstly, while there have been many attempts over the past decades, no uniform approach on how to best measure productivity has been made. The variety of

approaches is also an important topic in research, with papers reviewing available literature or pointing out the divide in research (Chapetta & Travassos, 2016; Forsgren et al., 2021; Wagner & Ruhe, 2018).

Approaches differ mostly in their use of data artifacts to determine productivity. There are many different aspects of a software developer's work that can be representative of productivity. However, determining which of these artifacts are relevant and useful for creating a productivity metric is a difficult task. de Aquino Junior and de Lemos Meira (2009) for example classify the artifacts generated from the software engineering process into:

- Physical Size: Use the physical size of a contribution (e.g. SLOC, created files)
- Design Size: Use the design of a contribution (e.g. number of classes and modules in the code)
- Functional Size: Use the number of features of a contribution.
- Value Size: Use the estimated added value of a contribution.

Overall, it is easier to measure physically based indicators, as they can be extracted directly from the source code. Thus, more recently, Treude et al. (2015) and Murphy-Hill et al. (2019) divided the topic into objective and subjective productivity metrics. Objective approaches consider components that can be directly measured from the output of the work (e.g. SLOC, commits, files). Subjective approaches are more focused on measuring personal evaluations that are more difficult to determine (e.g. qualitative surveys of developers). Table 2.1 gives an overview of examples of objective and subjective approaches used in productivity research.

Table 2.1: Comparison between objective and subjective approaches in software engineering literature

Objective Approaches	**Subjective Approaches**
Parizi et al., 2018: Analyses software engineering productivity by measuring contributions in git	**Murphy-Hill et al., 2019**: Analyses software engineering productivity by surveying developers about their experience
Gousios et al., 2008: Analyses software productivity by measuring developers contribution to the project	**Meyer et al., 2014**: Analyses software productivity by surveying developers about their perceptions of productivity

In this thesis we use this classification and focus on creating an objective approach, because we want our approach to be easily usable on any repository

using Git and GitHub. The data found in the metadata of these data sources is objective. We do however consider the insight gained from subjective approaches if they can be translated to an objective approach.

Besides the artifacts used for a productivity metric, one must also consider the source of such artifacts. The software engineering process produces a high amount of artifacts that can be difficult to keep a track of. Overall, there can be more to a software engineering project than the software code itself. Documentation, discussion and planning are examples of this (Diamantopoulos et al., 2020; Meyer et al., 2014). As a result, Murphy-Hill et al. (2019) explain that simplistic approaches only considering SLOC will not be sufficient for a metric. Additionally, some factors such as meetings or planning can be difficult to measure, because it is not immediately apparent what the indicators are Treude et al., 2015.

A problem we encountered while planning the development of this thesis, is the variety of tools used in software development. There are many different tools available to aid software development. As a result, it is likely that making a direct comparison between two software projects is challenging. Different software projects can use differing software tools (e.g. different bug tracking tools), that provide differing metadata on the development status.

2.1.3 Previous Approaches to Measuring Productivity

The topic of measuring productivity in software engineering is a chiefly studied topic in research. There have been a variety of approaches spanning multiple decades. In the following section, we give an overview of previous objective approaches split between earlier approaches and more recent Git based approaches.

Basic approaches

One common approach used early in productivity research is the use of SLOC. For this approach, the developer's contribution to the code is measured by their direct contribution in added lines of code. While analyzing previous literature on the inputs and outputs used for productivity metrics, Hernández-López et al. (2015) for example found that the most common productivity metric is defined as the ratio between SLOC and effort. Effort is commonly measured by the time developers spend on the work. The prevalence of the SLOC based objective productivity metrics is mentioned commonly in literature (Chapetta & Travassos, 2016; de Aquino Junior & de Lemos Meira, 2009; Oliveira et al., 2020). However, as previously mentioned, it is generally considered an insufficient metric for productivity. de Aquino Junior and de Lemos Meira (2009) for example state that the output of the software engineering process is more than just SLOC. (Oliveira et al., 2016) describe the ease of obtaining the relevant data as one of the main factors for its popularity.

git-based approaches

More recently, software engineering productivity approaches have started using Git to measure objective productivity indicators. Git is a version control system that is used as an archive for the development process. Each code contribution is stored and provides a variety of metadata about the contributions, such as added lines of code, number of commits and the exact time of the contributions. Many of the more recent approaches are based on the data collected this way.

Gousios et al. (2008) for example extracted repository data to determine actions that developers partake in during the development process. This encompasses actions such as added SLOC and commits of new files. From this data, they measure how much each developer contributes to the source code. Similarly, Parizi et al. (2018) used this to extract information about commits, SLOC, performed merges, added files and the time spent on the work. From this data they then measure individual developer performance. Furthermore, Hamer et al. (2021) use commits, merges and churns (lines added and removed) to measure contributions in software projects.

Another indicator used is the issues feature. Treude et al. (2015) for example describe commit messages and issues as a potential summary source for the work a developer performs. While surveying 156 developers, they found that both issues and commits are commonly regarded as objective measures for activity. Diamantopoulos et al. (2020) also further developed the use of git, by adding online repository data about issues collected on GitHub.

2.2 Inner Source

A dedicated approach to software engineering is often a closed and centralized process. Organizations split their software developers into departments and have them work on a singular project Raymond (1999). However, with the success of open source, organizations have begun to translate and apply the more distributed process to their own software engineering process. This use of open source development methodology in companies is called *inner source*.

Stol et al. (2014), describe a transparent and universally accessible setup of the code as some key practices used in an inner source approach. This is comparable to open source projects being hosted on public repositories for everyone to observe and contribute. In the case of inner source this is restricted to the internal use within the company. The goal of this is for software developers to be able to freely contribute to any project in the company. This is designed to reduce barriers that might hinder a software developer work and in return improve volunteering, collaboration, task self-selection and a faster development process among other things (Capraro & Riehle, 2016; Morgan et al., 2011; Stol et al., 2011).

2.3 Productivity Measurement in Inner Source

While the topic of measuring software engineering productivity has been an important research topic, measuring inner source productivity has not received the same attention. As far as we can tell, no research into the matter has taken place so far. However, being able to determine the value of conducting an inner source approach is important for both research and practice. It could for example be used to enable a direct comparison between the dedicated approach and inner source approaches and in return give insight on which development model to practice in the future. Thus, the lack of a software engineering metric that can measure inner source and the dedicated approach is important to solve.

3 Objective Definition

In the previous chapter we found that so far, there is no universally accepted solution to measuring software engineering productivity. Furthermore, we found no approach to measuring inner source productivity. As a result, we want to define a metric that measures software engineering productivity and is compatible with inner source. For this, we have chosen six criteria we want our solution to fulfill.

Firstly, as described in the previous chapter, there is large amounts of literature that doesn't necessarily correlate. Thus, we want our solution to use previous literature as a basis. More specifically, we want our solution to compile information from previous approaches, in order to find commonly used metric components. Our findings from literature should then be used to create our metric.

Secondly and thirdly, the metric we create should be able to measure software engineering productivity in the dedicated approach, as well as software engineering productivity for the inner source approach. This means the centralized approach to software engineering practiced for a long time and the more recent open source based approach.

Fourthly, based on the previous two points, we want our metric to enable making a direct comparison between the two approaches to software engineering.

Fifthly, we have to consider the feasibility of our approach. Any metric we create will be limited by the available data we can compose our metric off. As there are a near limitless number of ways to write software or to contribute to the software engineering process, we have to limit the ones we use in this thesis. Thus, we want our solution to consist of repository data available on GitHub[1]. GitHub was chosen, because it is a commonly used and accepted code hosting platform, that gives access to Git metadata and GitHub metadata (e.g. SLOC, developer communication and exact time points). The information we can extract from these data sources will be objective.

Lastly, we want the output of our metric to be easily reused and visualized. For this an appropriate data format, as well as a suitable representation is required.

[1]GitHub - https://github.com/

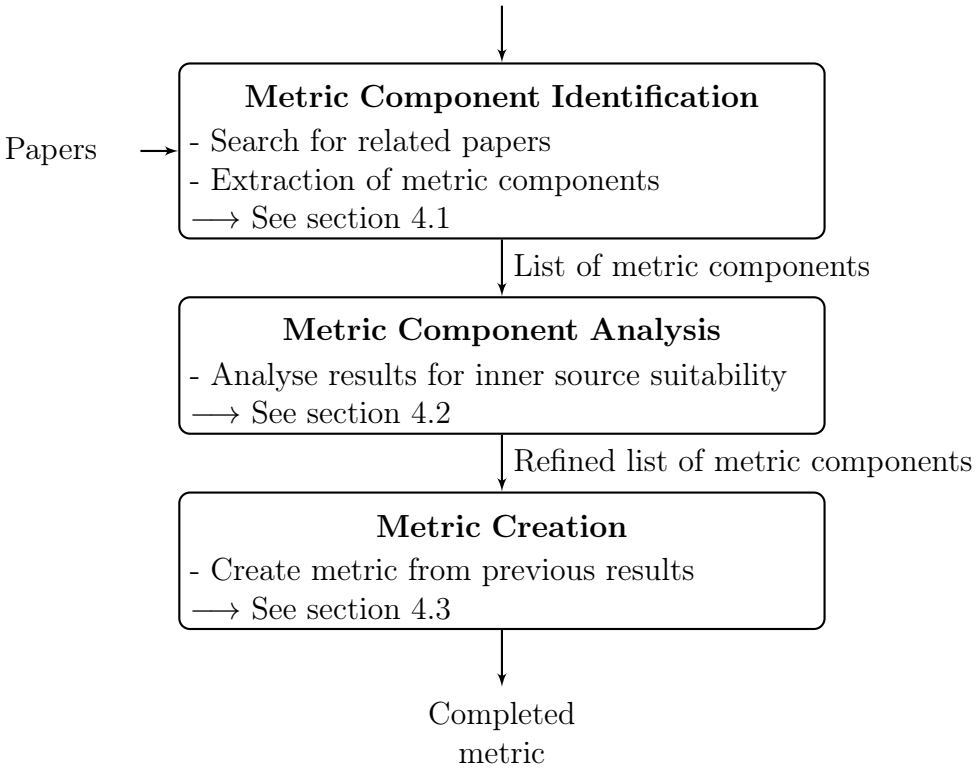
To summarize, the software engineering productivity metric created in this thesis must fulfill the following criteria:

1. The metric should take the existing body of literature on software engineering productivity into account.
2. The metric should be usable to measure software developer productivity in a dedicated software development environment.
3. The metric should be usable to measure software developer productivity in an inner source environment.
4. The metric should enable the comparison between an inner source approach and a classic software development approach.
5. The metric should make use of objective data openly available in GitHub repositories.
6. The metric should store it's output in a reusable and visualizable format.

4 Solution design

For the construction of our metric, we will firstly consider the previous work on the subject in-depth. We will analyze a selection of papers (see subsection 4.1.1 for selection process) for the metric components the individual metrics are composed of. From this data, we will count the rate of occurrence of each metric component. Afterwards, we analyze the suitability of the found metric components for inner source software engineering. From the gathered data of these tasks, we will then describe and create our metric approach. Figure 4.1 gives a detailed overview of the process.

Figure 4.1: Overview of the solution design process



4.1 Identifying Metric Components

In order to create a metric, we must first determine which objective components our metric will be composed of. As there are many different data sources available, we base our choice of metric components on previous literature. For this process, we will first detail how we selected the papers included in our analysis. Secondly, we will give our analysis of the metric components found.

4.1.1 Approach to Finding Papers

In order to find suitable papers describing the measurement of productivity in software engineering, we searched for papers using Google Scholar. For this process, the following key search terms were used:

- Software engineering productivity
- Software engineering performance
- Software developer productivity
- Software developer performance

Papers were accepted if they fulfilled one of the following criteria:

1. The paper deals with the creation of an objective software engineering productivity metric.
2. The paper deals with the creation of an objective software engineering metric that matches our definition of productivity.
3. The paper analyzes factors that objectively contribute to software engineering productivity.
4. The paper makes findings about objective software engineering metrics in subjective research.

4.1.2 Metric Component Identification

Within the papers found in subsection 4.1.1, we analyzed the information, in order to extract the components used to create the metrics. In the following, a metric component refers to a specific

.Occurrences of a specific metric component were counted and summed up. For a selection criteria, we used our previous objective definition. Thus, we only included metric components that could feasibly be implemented with Git and GitHub data. Furthermore, metric components with less than 4 occurrences will not be used for the creation of our metric. We summed up metric components in categories, if they were similar to each other (e.g. lines added and lines removed

were counted under SLOC, as they were usually used in combination). Overall, we found 8 different components and 30 total mentions. Table 4.1 gives an overview of each found metric component, as well as the papers it was used in and the total sum of occurrences.

Table 4.1: Overview of identified metric components

Name	**Description**	**Authors**	**Sum**
Commits	Describes the number of commits a developer contributed to the code. This can for example be counted for a specific timeframe.	Gousios et al., 2008, Oliveira et al., 2020, Hamer et al., 2021, Parizi et al., 2018, Treude et al., 2015, Diamantopoulos et al., 2020	6
SLOC	Describes the number of lines a developer added to the code. We also include lines removed and changed to accommodate for refactoring. This can for example be counted per commit or for a specific timeframe.	Gousios et al., 2008, Oliveira et al., 2020, Hamer et al., 2021, Parizi et al., 2018, Treude et al., 2015, Diamantopoulos et al., 2020, MacCormack et al., 2003, Chapetta and Travassos, 2020	8
Files	Describes the number of files a developer has commited to the code. This is usually counted for a specific timeframe.	Parizi et al., 2018, Diamantopoulos et al., 2020	2
Pulls	Describes the number of pull requests the developer is responsible for. This is usually counted for a specific timeframe.	Treude et al., 2015	1

Table 4.1 – *Continued from previous page*

Name	Description	Authors	Sum
Merges	Describes the number of lines a developer added to the code. This is usually counted for a specific timeframe.	Hamer et al., 2021, Parizi et al., 2018	2
Bugfixing activity	Describes the developers contribution to bugfixing activity. This can for example be partaking in discussions or commits associated with a solved issue.	Gousios et al., 2008, Treude et al., 2015, Diamantopoulos et al., 2020, Meyer et al., 2014	4
Work time calculations	Describes the estimated work time of a developer.	Parizi et al., 2018, Treude et al., 2015, Diamantopoulos et al., 2020	3
Community activity	Describes the developers contribution in discussions surrounding the development. This can for example be contributions to documentation.	Gousios et al., 2008, Treude et al., 2015, Chapetta and Travassos, 2020, Meyer et al., 2014	4

As can be seen, SLOC has been the most commonly used metric component. This is consistent with the literature described in chapter 2. Furthermore, commits are a commonly used metric component. This is also consistent with the literature we analyzed. Bugfixing activity and community activity are used less frequently than the other two, but still pass the threshold we set. Bugfixing activity is based on the actions a developer takes to fix errors in the code. Community activity refers to documentation work.

According to our selection criteria for metric components, we will create our metric using SLOC, Commits, bugfixing activity and community activity.

4.2 Evaluating Metric Components

After identifying the different metric components in the last section, we will now analyze each found metric component for suitability of measuring inner source software engineering productivity. To do so, we analyzed our description of inner source from section 2.2. From this, we defined two attributes commonly associ-

ated with inner source software development to be used as selection criteria. The metric components must fulfill both criteria for us to consider them during the creation of our metric. We define the selection criteria as:

In order to represent inner source software development any metric component must fulfill the following two criteria:

1. A contribution to an inner source software engineering project must capture the flow of the contribution across department boundaries.
2. A contribution to an inner source software engineering project must represent the high frequency of the contribution.

Criteria 1 was chosen, because the cross department structure is central to inner source development. Thus, a contribution made to the development process must be one that can measure this process. For example, adding a SLOC is a contribution that can be from a single department and from several different departments. As such, this would be considered as fulfilling criteria 1.

Criteria 2 was chosen, because being able to add to a software project at time is important to the nature of inner source. For example, adding a SLOC is a contribution that can be added at any moment in time and is not restricted to any specific time. As such, this would be considered as fulfilling criteria 2.

Table 4.2 shows our evaluation of each metric component. All components found fulfilled both criteria. Overall, the objective nature of our data sources harmonizes well with inner source. Contributions can be made by any developer in the company. As a result, this represents the cross department structure of inner source well. Furthermore, the contributions are also high frequency, as there is no specific timeframe to making a contribution.

Table 4.2: Evaluation of inner source suitability of each metric component

Metric Component	**Criteria 1**	**Criteria 2**
Commits	✓	✓
SLOC	✓	✓
Bug fixing activity	✓	✓
Community activity	✓	✓

4.3 Metric Creation

In the previous chapter, we analyzed what metric components represent an inner source software engineering productivity metric. In the following, we will describe how we construct our metric from the information learned. For this we will first

describe the dimensions of our metric. Then we will detail how we weigh the different metric components. Lastly, we will describe the completed metric.

4.3.1 Metric Basics

Our metric is designed to deal with the objectives defined in chapter 3. The following section describes some aspects we want to clarify before giving a complete definition.

Value Range

Firstly, we wanted to ensure that point 4 of our objective definition is fulfilled. For this we have chosen to make our metric ratio scaled in a range of 0-1. As a result, two software repositories can be compared, because the determined values are always in the same range of values.

Determining metric component share

The created metric will determine the weight of each metric component by using the results from chapter section 4.2. Thus, we will award each identified metric component with points based on the number of occurrences in literature. For this we have chosen a tier based system. Overall, there are three tiers, with the first tier being reserved for the most common occurring metric component. The subsequent tiers are designed to hold metric components with a medium and small number occurrences. Table 4.3 gives an overview of the classification into tiers for each metric component.

Table 4.3: Classification of metric components into tiers based on number of occurences

Tier	Range	Share	Metric Components
A (Most common)	[10 - 8]	5 Points	SLOC
B (Medium common)	[7 - 5]	4 Points	Commits
C (Least common)	[5 - 3]	3 Points	Community Activity
			Bugfixing

The point based system is used to ensure that metric components represent a customizable fraction of the total value. This counteracts the differing availability of software tools. As previously stated, there are a lot of different tools available for supporting the development process. Thus, a metric must be adjustable based on what tools are available in each instance. An implementation can use binary values (0: Not available, 1: Available) to set metric components according to availability. If a software project for example does not use GitHub Issues

(metadata associated with bugfixing), the bugfixing activity component can be neglected by excluding the metric component from the configuration. As the share is based on the total points. the metric automatically adjusts. Figure 4.2 and Figure 4.3 give an example of how the system works with two and three metric components respectively.

Figure 4.2: Example for the point based system for metric component share with two components

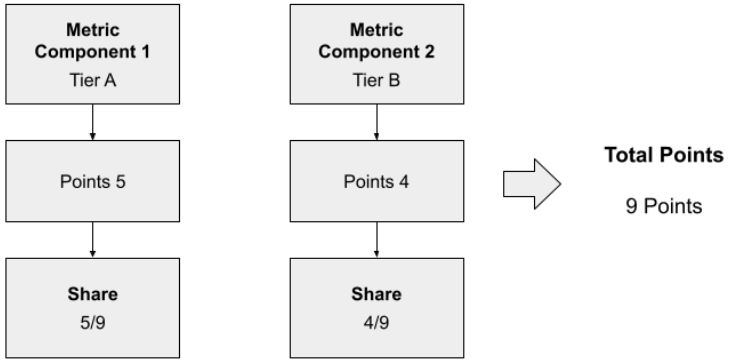
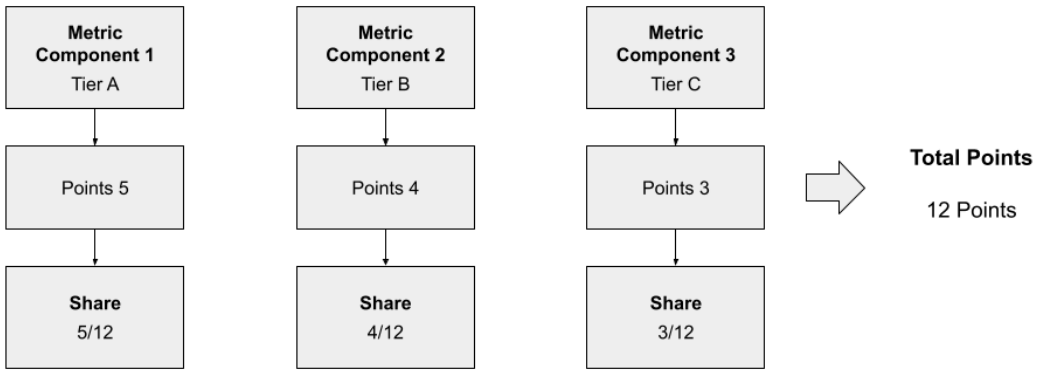


Figure 4.3: Example for the point based system for metric component share with three components



As can be seen in the examples, the metric will adjust automatically based on the number of metric components. As there are only two metric components in Figure 4.2, the overall share for each metric component is larger. This can also be used for adding more values later, in order to expand upon the work in this thesis.

4.3.2 Metric Description

We define our metric for software engineering productivity as the sum of each metric components individual evaluation:

$$Productivity = \sum_{n=1}^{\#-Metric\ Components} Metric\ Component\ Evaluation_n \tag{4.1}$$

The Metric component evaluation is determined for each metric component individually. It is calculated, by evaluating the developer's performance in corresponding and relevant metadata available from a chosen data sources (e.g. Git and GitHub in this thesis). For example, the metric component SLOC could be determined by the amount of lines added and removed during development. The final value is defined as the average of each metadata value:

$$Metric\,Component\ Evaluation = \frac{\sum_{n=1}^{\#-Metadata\ components} Metadata\ component_n}{\#-Metadata\ components} \tag{4.2}$$

The individual metadata values are determined by splitting the total share of the metric component into different value ranges. For this, a maximum value is set and the value ranges are split into even parts based on this value. The maximum value is set by the user and described in more detail in subsection 4.3.3. Based on the performance in these categories we then assign the performance of a developer in each metadata component to a specific value range. Each of the value ranges is associate with an evaluation result, that is calculated by splitting the overall share of the metric component by the same number as the value ranges. Thus the final result for each metadata component cannot exceed the evaluation of the metric component.

$$Metadata\ component(x) = \begin{cases} 0 & if\ x \in [0; y_1(\\ ... & ... \\ Share & if\ x \in (y_{n-1}; \infty] \end{cases} \tag{4.3}$$

with:

x: Objective metadata component contribution of a developer

$y_1, ..., y_n$*: Borders of the value ranges*

Take for example a metric component *Commits* with:

- Total number of commits as the only metadata component
- A 3/10 share
- A maximum value of 400

This could for example be split into 4 value ranges:

$$Number\ of\ Commits(x) = \begin{cases} 0 & if\ x \in [0; 100(\\ 0.1 & if\ x \in (100; 200(\\ 0.2 & if\ x \in (200; 300(\\ 0.3 & if\ x \in (300; \infty] \end{cases} \tag{4.4}$$

Thus, a developer that added 242 SLOC in a specific timeframe would receive an evaluation of 0.2. As this is the only metadata component, this would also be the evaluation for the metric component itself.

As can be seen in this example, by setting the maximum value to 400, we can rule out any outlier results for a metadata component. This could for example be library imports with an irregularly high number of SLOC.

4.3.3 Weighing

As there are many different components in our metric, that don't necessarily have the same range of values, we need to scale the values. It is for example not possible to simply take commits and lines added and make a direct comparison. In this scenario, lines added would eclipse commits, as one commit is not equal to one line added. Thus, we need to properly scale these values before using our metric. Finding a perfect measure for this is difficult, as this can vary depending on the software project at hand. For this thesis, we calculate the value, by dividing the total number for all contributions of a metadata component by the total number of developers.

5 Implementation

This section goes over the implementation of the solution described in chapter 4. Firstly, we will cover different tools that were used during the development process. Secondly, we will describe the data sources and the data collection process. Lastly, we describe the identification of suitable metadata values and the compilation of the data for our metric.

5.1 Used Tools

For the implementation of our solution we require a variety of software tools. Firstly, we make use of GrimoireLab, a toolset for software development analytics, to extract and compile data. Secondly, we describe the Application Programming Interface (API) used for data collection. Lastly, we use Kibana, an analytics and search dashboard, for query generation.

5.1.1 GrimoireLab

To aid our implementation, we used the software tool GrimoireLab[1]. GrimoireLab was originally designed as a research project and is now maintained by the CHAOSS[2] project (Dueñas et al., 2021). The tool's described use case is for software development analytics and offers a variety of algorithms to extract data from different sources (e.g. Git, GitHub). The following features make it relevant for this thesis:

- **Data extraction**: In order to implement the metric defined in section 4, we require a way to continuously extract relevant data from a variety of data sources. Most importantly, we need to access repository data (e.g. commits and dates), as well as GitHub related data (e.g. GitHub Issues). GrimoireLab offers an implementation to extract and GitHub specific data.

[1]GrimoireLab - https://github.com/chaoss/grimoirelab
[2]CHAOSS - https://chaoss.community/

- **Data compilation**: In order to compile the metric from its different components, we need to store the collected information in an easily retrievable format. GrimoireLab stores all retrieved data in an Elasticsearch database. Thus, we have a way to store and retrieve data. Furthermore, we can add additional data not supported by GrimoireLab at a later point (See section 5.2 for more information).

Overall, GrimoireLabs offering is quite extensive, however we add some additional data required for our solution. This is described in section 5.2.

Components

The overall GrimoireLab application uses a variety of tools. Each tool is its own software repository and is responsible for a different task in the overall process. We will make use of multiple components, but specifically focus on sirmordred[3], elk[4] and Perceval[5].

sirmordred Sirmordred is the component used to coordinate the different grimoirelab tools. It configures the basic setup and determines which repositories are analysed. In this thesis we will use it to set the analyzed repositories to the configuration.

Perceval Perceval is the component handling the data gathering process by collecting raw data from specified sources. For this thesis we will use it to gather data from and GitHub data sources.

GrimoireElk GrimoireElk is the component responsible for handling the ElastiSearch database connectivity. Any data extracted in the data gathering process will be processed here and added to the database. In this thesis, we will use it for storing our data.

5.1.2 Elasticsearch and Kibana

For data storage we use Elasticsearch[6]. Elaticsearch is the database used for storing data in the GrimoireLab process. Data is added by GrimoireElk and can then be visualized by Kibana.

Kibana[7] is a 'browser-based analytics and search dashboard'. GrimoireLab offers

[3]sirmordred - https://github.com/chaoss/grimoirelab-sirmordred
[4]elk - https://github.com/chaoss/grimoirelab-elk
[5]Perceval - https://github.com/chaoss/grimoirelab-perceval
[6]Elasticsearch - https://github.com/elastic/elasticsearch
[7]Kibana - https://github.com/elastic/kibana

Kibiter[8], a custom version of Kibana, that is designed to directly work with it's own tools. For this thesis, we will use the tool to generate predefined queries for our data compilation scripts (see section 5.4).

5.2 Data Sources

For data sources we use and GitHub as defined in chapter 3. GrimoireLab's data extraction feature allows us access to predefined schemata for both. The schemata define a set of metadata values (e.g. lines_added, author_name etc.) for data source that can be used to find relevant information. This process is handled by Perceval and GrimoireElk. Firstly, raw data is extracted by Perceval. For Git, this is for example information about the developer responsible for the commit or the size. This information is found within the system. Raw data from GitHub is accessed by API calls to different systems. For this thesis GitHub Issues is the most relevant data source for GitHub. As pull requests and issues are classified in a similar way in GitHub, we exclude information about pull requests and focus strictly on issues.

Secondly, repository metadata from is stored in Elasticsearch by commits as a unit. GitHub Issues metadata is stored by Issues. There are a total of 124 and 72 values associated with each respectively. These values are partially extracted directly from repositories and GitHub Issues. However, some values are calculated during the GrimoireLab workflow. This includes internal identifiers for authors and organizations, as well as information about when the data was extracted. Thus, not every value can provide relevant metadata for our metric and we will only analyze relevant values (See section 5.3 for our choice of values).

Furthermore, we noticed that the github_issues values only lists the author of an issue. For our metric we also wanted to include developers that contribute to the discussion surrounding an issue or contribute by writing code. Thus, we implemented a script that adds an *involved* value listing all contributors to an issue that also have at least one commit. Developers involved in issues that do not have a commit cannot be queried and are ignored in the following. This data is extracted with the python script `update.py` and added to the Elasticsearch database. The process is done in three steps:

1. Query Elasticsearch for a list of every developer that has contributed to the code.
2. For each developer, query the GitHub Search API for a list of all GitHub Issues the developer was involved with and keep a dictionary for each issue.
   (a) The request is made by using `curl` and customizing:

[8]Kibiter - https://github.com/chaoss/grimoirelab-kibiter

https://api.github.com/search/
issues?q=repo:{repo_name} 20involves:{developer_account}

(b) The data is limited to 30 requests per minute and must be requested periodically.

3. For each issue, write the involved value into the database

Afterwards, we create a scripted field in Kibiter that combines the involved values and the developer that authored the issue (If not already present in the involved value). Scripted fields are a feature of Kibana an allow for the creation of new data tags without the need of adding to the code base.

5.3 Identifying Metric Component Metadata

In chapter 4, we described the different metric components our metric is going to be composed of. As there are many ways the metric components can be represented, we will offer default values that our implementation will use. However, the code is implemented in a way to allow further additions depending on availability. This section will describe the mapping of each metric component to the corresponding default values in the data sources. We approached this process as follows:

1. Analyze the available data sources for fitting metric component metadata.
2. Compile a list for each metric component, containing all possibly relevant metadata (See Table 5.1).
3. Analyze the list based on feasibility and suitability.

Table 5.1: Relevant git and GitHub values for each metric component

Metric Component	**Data Source**
Commits	Count the total commits associated with a developer.
SLOC	git: lines_added git: lines_removed git: lines_changed
Bugfixing activity	github_issues: involved github_issues: time_to_close_days github_issues: time_to_first_attention
Community Activity	Analysis done on the documentation repository.

For commits we can count the total number of commits. We don't differentiate between the sizes of the commits, because SLOC covers this aspect. SLOC on the

other hand provides three values that are associated with the metric component. However, for our implementation we will only use lines_changed, because it combines lines_added and lines_removed. Thus, including all would result in evaluating values multiple times. For bugfixing activity, we make use of the GitHub Issues dashboard of GrimoireLab. Relevant values include the involved value described in the previous chapter, time_to_close_days and time_to_first_attention. From these values we have decided to solely focus on the involved value. Community activity is represented by the work developers make on the documentation. Thus, as documentation for large repositories is often stored in and GitHub, we will do our analysis on the corresponding repositories.

5.4 Compiling Metric Component Metadata

After gathering all data in the Elasticsearch database, we parse and compile the data according to our metric definition in section 4.3. For this we will first describe the configuration file that allows the implementation to be customized and then detail the actual implementation of our metric.

5.4.1 Configuration

While designing our implementation, we paid attention to making it extendable and customizable. This will allow us to add more tools later and to adjust values depending on the project at hand. To achieve this we created a `config.ini` file. This file is parsed upon starting the python script and contains configuration values that can be edited by a user without having to adjust the code. Listing 5.1 gives a compressed overview of the relevant sections.

Listing 5.1: Compressed overview of the config.ini file

```
[Basic]
metric_components = Commits, SLOC, Bugfixing, Community
tier_weights = {"A":5,"B":4,"C":3}
split = 4
repo = https://github.com/...
start_date = 2022-01-01 00:00:00.000
end_date = 2022-12-31 23:59:59.999

[SLOC]
tier = A
metadata = lines_changed
metadata_max = 100
```

The values under `[Basic]` are used to define the core functionality of the metric:

- `metric_components`: Defines what metric components will be used. This is used to allow customization depending on the individual situation.
- `tier_weights`: Defines how many points are awarded for each tier.
- `split`: Defines how many value ranges there are for each metadata source.
- `repo`: The repository to be analyzed.
- `start_date` and `end_date`: Defines the time span of the analysis.

The values under `[Commits]` are used to define how to handle the individual metric components (Commits is used as an exemplary value, each other metric component has their own section):

- `tier`: Defines which tier the metric component belongs to.
- `metadata`: A list that defines which tags and data sources the metric component consists of.
- `metadata_max`: A list that defines the maximum value of each tag listed in metadata. This value is then used to split the evaluation of each tag into split ranges of values.

5.4.2 Data Processing

The actual python scripts for the data compilation are coordinated by a `main.py` file that calls different functions. The file has five tasks:

1. Parse all relevant fields from the `config.ini` file.
2. Establish a connection to the Elasticsearch database.
3. Determine the total points by summing up all individual metric component values. This allows us to determine the share of each metric component by calculating: $\frac{individualpoints}{totalpoints}$.
4. Calculate the individual metric component evaluation by using Equation 4.2 and Equation 4.3. A specific `evaluate`-function is called for each metric component. The evaluation of community activity is handled by executing the main process on the documentation repository.
5. Compile the previous results by using Equation 4.1.

Calculating individual metric component evaluation

The individual metric component evaluation functions are responsible for querying the Elasticsearch database for all relevant values gather in section 5.3. The functions are structured as follows:

Listing 5.2: Function definition of the compile_data function

```
def evaluate_sloc(share: float, metadata: List[str],
    metadata_max: List[str], repo: str, es, start_date: str
    , end_date: str) -> dict:
```

The function arguments are defined as follows:

- `share`: Describes the share of the final result.
- `metadata`: A list of which values need to be extracted.
- `metadata_max`: The maximum value corresponding to each metadata value.
- `repo`: The link to the repository to be analyzed.
- `es`: The Elasticsearch connection.
- `start_date` / `end_date`: Used to determine the range of dates for the final output.

We query the Elasticsearch database for each metadata value specified for the metric components. This is done by generating an Elasticsearch query from preset requests generated in Kibiter. The values for dates, repository name and metadata value can be adjusted depending on the arguments in `metadata`, `start_date` and `end_date`. The response represents the raw results for individual metadata components (e.g. total number of lines_added). This value is then allocated to a value range as defined in Equation 4.3. For this we divide the `metadata_max` and `share` values by `split`. From these values we can generate the value ranges and their corresponding metadata component evaluation value. The final values are saved for each developer in a dictionary and returned to the main process for the final output.

5.5 Storing Output

After the data processing is finished, the `main.py` script writes all data to a Comma-separated values (CSV) file. This format saves the data for each metric component in a line. Each line lists the individual values of each developer separated by a comma. This allows users to store the data separately from the code in a format that can be transformed to other formats with ease.

6 Demonstration

The following section will demonstrate the functionality of our implementation by analyzing a data sample. For this we will describe the data sample chosen, the configuration of our solution and the detailed results.

6.1 Data Sample

For the data sample we chose repositories involved with the core GrimoireLab project. This sample consists of 8 repositories with a total of 10.906 commits and 1164 GitHub issues from November 3rd 2017 to December 8th 2022. All data used in this sample was obtained from publicly available GitHub repositories. A complete list of each repository included in the data set can be found in Appendix A.

The data sample was chosen, because we wanted to show the functionality of our developed solution on a real world example. The structure of the repositories is that of an open source project. However, it is also representative of attributes common to regular software engineering and inner source. All contributions are made by commit and issues are logged. Furthermore, open source projects also have cross boundary contributions that are open to everyone, as well as high frequency contributions that can be made at any time.

The data was gathered on December 8 2022 using the default docker-compose setup as recommended in the GrimoireLab repository. For this we used the 0.7.0 release of GrimoireLab and the 6.8.6 version of Elasticsearch.

6.2 Configuration

For the basic configuration we determined the initial maximum values by analyzing the entire data sample. We determined the average of each value by dividing the total amount in our sample through the number of developers. For SLOC we for example counted the total amount changed by all developers and then divided the number by the total amount of developers. Some small adjustments

were made to accommodate outliers, such as the import of libraries, as they would falsify the end result. Accordingly, the values for the overall evaluation are set as follows:

- `[SLOC] metadata_max`: 2611
- `[Commits] metadata_max`: 29
- `[Bugfixing] metadata_max`: 3
- `[Community] metadata_max`: 1
  - `[SLOC] metadata_max`: 2104
  - `[Commits] metadata_max`: 8
  - `[Bugfixing] metadata_max`: 6

We then repeated this process for each repository, in order to obtain values for individual comparisons in the next section. Table 6.1 lists all values used for each repository:

Table 6.1: Classification of metric components into tiers based on number of occurences

Repository	SLOC	Commits	Bugfixing
grimoirelab	300	4	14
grimoirelab-graal	255	6	1
grimoirelab-perceval	669	13	3
grimoirelab-elk	1085	20	2
grimoirelab-sortinghat	2412	18	4
grimoirelab-sirmordred	395	11	2
grimoirelab-kibiter	5104	36	1
grimoirelab-sigils	591	12	1

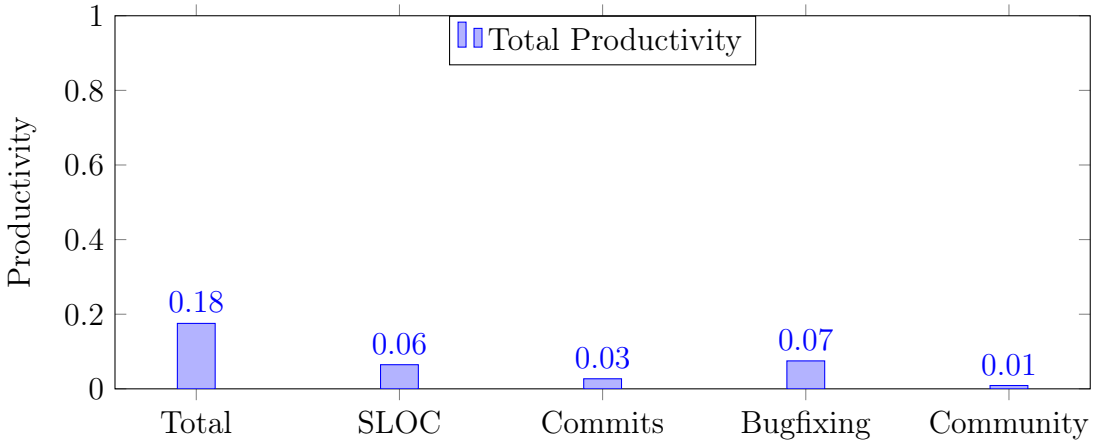
6.3 Results

6.3.1 Overview

Firstly, we wanted to demonstrate the overall functionality of our solution, by giving an overview of the productivity metric for the entire data sample. This is achieved by listing all repositories in the `repo` variable of the configuration. The results can be seen in Figure 6.1.

From this, we noticed that one issue with our metric is the abundance of developers that have only a small amount of contributions. This can be attributed

Figure 6.1: Average productivity for each metric component for all repositories combined



to the repositories being open source. We checked a small sample of developers with low contribution by hand and noticed that they were either small changes or developers creating a single issue to ask for help. As a result, the final evaluation pictured in Figure 6.1 is warped. Figure 6.2 gives an overview of the frequency of developers divided into value ranges from (0.0-0.1) to [0.9-1.0]. As can be seen, in our sample, there are 101 developers that have an evaluation of 0. We argue that this is to be expected, as our implementation takes into account that there are multiple actions a developer can conduct. Opening one issue is unlikely to increase the productivity of the overall project. For the evaluation in this chapter, we will exclude all developers that have an evaluation of 0. Figure 6.3 shows the average evaluation with the adjustments.

The frequency of developers shows that there is a split between developers in the repositories. There are few developers that have a high productivity, while a larger amount is showing lower productivity. This is consistent with the repositories being an open source project. There are a few main contributors and other developers of varying productivity. The average productivity is also in line with this.

6.3.2 Comparison

To analyze the sample further, we used our implementation to generate the results for each repository individually. To do this we can change the `repo` value in our configuration and use the configuration values described in Table 6.1. From this we can enable a direct comparison of the individual results. An overview of the average productivity values can be seen in Table 6.2 and the number of developers per productivity interval can be seen in Table 6.3.

Figure 6.2: Number of developers per productivity interval

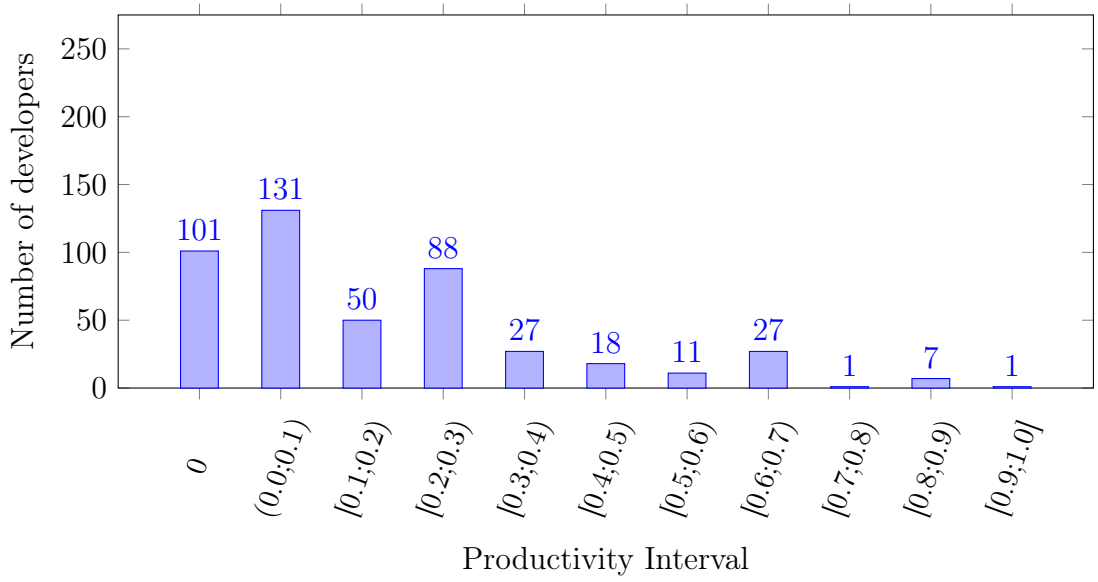


Figure 6.3: Average productivity for each metric component for all repositories combined with adjusted developers.

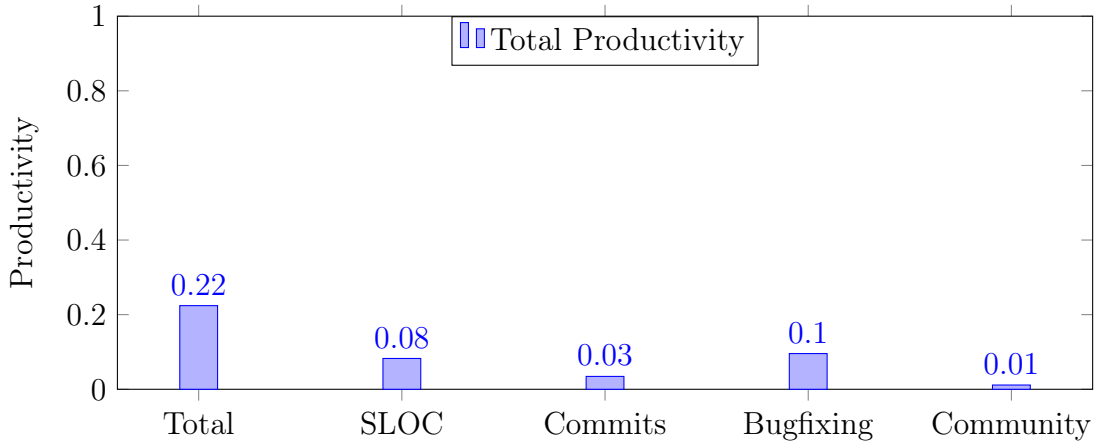


Table 6.2: Average productivity for each metric component for individual repositories

Repository	SLOC	Commits	Bug-fixing	Community	Total
grimoirelab	0.0619	0.01270	0.0419	0.0590	0.1756
grimoirelab-graal	0.03485	0.0091	0.0653	0.0721	0.1850
grimoirelab-perceval	0.0280	0.0051	0.0892	0.0297	0.1520
grimoirelab-elk	0.0493	0.0126	0.1170	0.0390	0.2179
grimoirelab-sortinghat	0.0390	0.0115	0.0684	0.05368	0.1726
grimoirelab-sirmordred	0.0497	0.0095	0.1210	0.0401	0.2203
grimoirelab-kibiter	0.1635	0.0752	0.0276	0.0336	0.2999
grimoirelab-sigils	0.0547	0.0159	0.0896	0.0617	0.2219

Table 6.3: Number of developers per productivity interval for each individual repository

	(0.0;0.1)	[0.1;0.2)	[0.2;0.3)	[0.3;0.4)	[0.4;0.5)	[0.5;0.6)	[0.6;0.7)	[0.7;0.8)	[0.8;0.9)	[0.9;1]
grimoirelab	39	14	4	6	1	1	2	1	1	0
graal	27	11	12	1	1	2	3	1	0	0
perceval	77	23	23	5	4	2	3	0	2	0
elk	20	35	35	3	1	2	4	2	4	0
sortinghat	42	13	12	1	1	4	1	1	1	1
sirmordred	20	30	32	7	3	4	4	1	2	0
kibiter	29	15	20	13	14	9	22	0	1	0
sigils	24	10	16	6	3	1	4	2	1	0

From this, we can see that while the numbers for all repositories are relatively close, some outperform the rest. For Kibiter, there is an over performance in SLOC, with the value being more than double that of the second largest repositories. Looking at the frequency of developers, one can notice a high number of developers in the (0.5;0.6] interval that is likely the cause for this number. GrimoireElk,Sigils and SirMordred follow Kibiter in average productivity. However, for these repositories, there is no clear value setting it apart from the others.

7 Evaluation

In this chapter, we will look back at the objectives defined in chapter 3 and evaluate whether our solution, implementation and demonstration fulfill the defined criteria. Furthermore, we will discuss limitations of our work.

7.1 Objective Criteria

For the evaluation, we want to look back at the six criteria we imposed on our solution in chapter 3. For this, we will compare each criteria to our findings in the previous chapters.

Firstly, as there are many existing approaches to measuring productivity, we wanted our solution to be based on the existing body of literature. To create our metric, we conducted a short review of existing literature on objective approaches to measuring software engineering productivity. This helped us base our metric on several metric components that represent a broad spectrum of the software development process. This exceeds the more basic approaches based solely on SLOC that have seen use in literature and practice.

$\Longrightarrow$ Criteria 1 is fulfilled.

Secondly and thirdly, we defined that our solution should be able to measure both the dedicated and the inner source software engineering approach. Our solution can be used on any repository that is hosted on GitHub. Furthermore, the solution is designed to be adjustable to other situations. Thus, the implementation could be built upon to include other means of gathering data. For our demonstration, we used an open source project, however, we argue that the data generated and collected is similar to the other two approaches. Thus, while we would like to conduct further analysis on this in the future, the criteria are still sufficiently fulfilled.

$\Longrightarrow$ Criteria 2 and 3 are fulfilled.

Fourthly, we defined that our solution should enable the comparison between the dedicated and inner source software engineering approach. For this we chose our

metric to be ratio scaled. This allows us to to make comparisons between average total values, developer distribution and more. Furthermore, the metric can be used on a single repository or multiples at once.

$\Longrightarrow$ Criteria 4 is fulfilled.

Fifthly, we defined that our solution should make use of objective data available in GitHub repositories. Our solution extracts all data from GitHub (Git repository, GitHub Issues, Search API).

$\Longrightarrow$ Criteria 5 is fulfilled.

Lastly, we defined that our solution should save the output in a format that is reusable and visualizable. Our implementation uses the CSV format, which can store the output in a separate file for later use. The format is also suitable for visualization, as it can be converted and used in the most popular data visualization software. Thus, this criteria is fulfilled.

$\Longrightarrow$ Criteria 6 is fulfilled.

7.2 Limitations

For this thesis, there are limitations we want to mention in this chapter. Firstly, by only focusing on objective factors we neglect a subsection of factors potentially influencing productivity. However, including more subjective factors would have exceeded the scope of this paper. Furthermore, relying on objective factors means that the system can potentially be deceived by developers intentionally producing an irregular amount of commits. This could lead to the results being skewed towards a higher productivity. Further research could improve on this by creating a more hybrid approach that considers more aspects of software development.

Lastly, we restricted the amount of data sources to data available on GitHub. This choice was made due to the scope of this thesis. However, for further research we would recommend expanding the scope to different data sources such as different bug trackers, communication tools and code hosting platforms.

8 Conclusions

This thesis analyzes ways of measuring software engineering productivity. To determine our own approach, we searched previous approaches for commonly used metric components. We found that SLOC, commits, bugfixing activity and community activity are the most frequently used.

From this, we created our own approach to measuring software engineering productivity. This approach scales each of the found metric component's relevance based on the frequency we found them in past literature. To gather data for our metric, we parse data available on GitHub. This data is compiled and the relevant information is extracted and summed up.

We also demonstrated the functionality of our approach, by analyzing a group of repositories belonging to a software project. This data showed that our approach can be used to determine productivity for a dedicated software engineering approach and an inner source approach.

For future research, we suggest expanding the scope of our approach to include more data sources. Furthermore, we think a hybrid approach between objective and subjective ways of measuring productivity could give more insight into productivity measurement.

Appendices

A List of Repositories used in Demonstration

A.1: Repositories used for demonstration

Repository
https://github.com/chaoss/grimoirelab
https://github.com/chaoss/grimoirelab-graal
https://github.com/chaoss/grimoirelab-perceval
https://github.com/chaoss/grimoirelab-elk
https://github.com/chaoss/grimoirelab-sortinghat
https://github.com/chaoss/grimoirelab-sirmordred
https://github.com/chaoss/grimoirelab-kibiter
https://github.com/chaoss/grimoirelab-sigils

References

Capraro, M. & Riehle, D. (2016). Inner source definition, benefits, and challenges. *ACM Computing Surveys (CSUR)*, *49*(4), 1–36.

Chapetta, W. A. & Travassos, G. H. (2016). Software productivity measurement and prediction methods: What can we tell about them? *Anais do XV Simpósio Brasileiro de Qualidade de Software*, 211–225.

Chapetta, W. A. & Travassos, G. H. (2020). Towards an evidence-based theoretical framework on factors influencing the software development productivity. *Empirical Software Engineering*, *25*(5), 3501–3543.

de Aquino Junior, G. S. & de Lemos Meira, S. R. (2009). Towards effective productivity measurement in software projects. *2009 Fourth International Conference on Software Engineering Advances*, 241–249.

Diamantopoulos, T., Papamichail, M. D., Karanikiotis, T., Chatzidimitriou, K. C. & Symeonidis, A. L. (2020). Employing contribution and quality metrics for quantifying the software development process. *Proceedings of the 17th International Conference on Mining Software Repositories*, 558–562.

Dueñas, S., Cosentino, V., Gonzalez-Barahona, J. M., del Castillo San Felix, A., Izquierdo-Cortazar, D., Cañas-Díaz, L. & Pérez García-Plaza, A. (2021). Grimoirelab: A toolset for software development analytics. *PeerJ Computer Science*, *7*(e601).

Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B. & Butler, J. (2021). The space of developer productivity: There's more to it than you think. *Queue*, *19*(1), 20–48.

Gousios, G., Kalliamvakou, E. & Spinellis, D. (2008). Measuring developer contribution from software repository data. *Proceedings of the 2008 international working conference on Mining software repositories*, 129–132.

Graziotin, D. & Fagerholm, F. (2019). Happiness and the productivity of software engineers. *Rethinking productivity in software engineering* (pp. 109–124). Springer.

Hamer, S., Quesada-López, C., Martınez, A. & Jenkins, M. (2021). Using git metrics to measure students' and teams' code contributions in software development projects. *CLEI Electronic Journal*, *24*(2).

Hernández-López, A., Colomo-Palacios, R., Soto-Acosta, P. & Lumberas, C. C. (2015). Productivity measurement in software engineering: A study of the inputs and the outputs. *International Journal of Information Technologies and Systems Approach (IJITSA)*, *8*(1), 46–68.

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, 75–105.

MacCormack, A., Kemerer, C. F., Cusumano, M. & Crandall, B. (2003). Trade-offs between productivity and quality in selecting software development practices. *Ieee Software*, *20*(5), 78–85.

Meyer, A. N., Fritz, T., Murphy, G. C. & Zimmermann, T. (2014). Software developers' perceptions of productivity. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 19–29.

Morgan, L., Feller, J. & Finnegan, P. (2011). Exploring inner source as a form of intra-organisational open innovation.

Murphy-Hill, E., Jaspan, C., Sadowski, C., Shepherd, D., Phillips, M., Winter, C., Knight, A., Smith, E. & Jorde, M. (2019). What predicts software developers' productivity? *IEEE Transactions on Software Engineering*, *47*(3), 582–594.

Oliveira, E., Conte, T., Cristo, M. & Mendes, E. (2016). Software project managers' perceptions of productivity factors: Findings from a qualitative study. *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*, 1–6.

Oliveira, E., Fernandes, E., Steinmacher, I., Cristo, M., Conte, T. & Garcia, A. (2020). Code and commit metrics of developer productivity: A study on team leaders perceptions. *Empirical Software Engineering*, *25*(4), 2519–2549.

Parizi, R. M., Spoletini, P. & Singh, A. (2018). Measuring team members' contributions in software engineering projects using git-driven technology. *2018 IEEE Frontiers in Education Conference (FIE)*, 1–5.

Peffers, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, *24*(3), 45–77.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, *12*(3), 23–49.

Stol, K.-J., Avgeriou, P., Babar, M. A., Lucas, Y. & Fitzgerald, B. (2014). Key factors for adopting inner source. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *23*(2), 1–35.

Stol, K.-J., Babar, M. A., Avgeriou, P. & Fitzgerald, B. (2011). A comparative study of challenges in integrating open source software and inner source software. *Information and Software Technology*, *53*(12), 1319–1336.

Treude, C., Figueira Filho, F. & Kulesza, U. (2015). Summarizing and measuring development activity. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 625–636.

Wagner, S. & Ruhe, M. (2018). A systematic review of productivity factors in software development. *arXiv preprint arXiv:1801.06475*.