Università degli studi di Catania

Dipartimento di Matematica e Informatica

XXXV Ph.D in computer science

Doctoral Thesis

Data Mining Techniques for Software Effort Estimation

Author:

Leonardo Pelonero

Supervisor:

Prof. Emiliano Tramontana

Uni ct MATEMATICA E INFORMATICA

Academic Year 2022/2023

Abstract

It is paramount to properly integrate effort estimation with good development practices. Effort estimation is an open challenges, and it is performed to prevent software defects and delays during the development. In agile projects, at the very beginning stage of the software development life cycle, correct effort estimation helps to determine the order of the tasks to perform. However, estimation can be complex and there is the risk of making erroneous estimations. Predicting the effort is important to validate the results obtained or monitor the progressive trend. While analogy and expert judgment based are popular effort estimation approaches, no tools exist to assist managers in verifying and supervising the progressive effort during project development, except for size measures such as Line Of Code (LOC) and Function Point (FP). Modern software repositories are a valuable support for large teams working together on a project. Besides the code, a repository includes useful data related to development practices as well as features that better identify the project's developers.

This thesis presents novel approaches and metrics based on data extracted from repositories and related to the software life cycle to profile developer activities. By acquiring employees log data, project management can be provided with useful information such as how much more effort it will take to complete, and how long that will take; however in open source repository such records do not usually exist. The value of the proposed metrics is to reveal the effort spent by developers, to estimate the additional effort required to complete the task, to highlight strengths and weaknesses, then suggest improvements. Furthermore, the proposed metrics can be easily integrated in the usual development environment, without affecting other practices. The above analysis can be performed both during the development process of new software components and features, or afterwards, when the application's development has been completed.

Contents

List of Tables

Chapter 1

Introduction

Software processes are constantly evolving as new and different technologies and applications are developed and used. However, while changes in software industry are arbitrary, the approaches to administer the software development remain almost unchanged. Thus, engineers and researchers in this sector attempt to provide useful and performing tools to support the maintenance of software systems, to improve the design and to promote software quality.

Software quality is driven by many aspects in software development. A primary goal of software development organisations is to initiate a project and finish it within acceptable schedule and budget [BTB07]. The competitiveness of software organisations depends on their ability to accurately predict the effort required and risks involved in developing software systems to avoid project abortion or restarts [SPH16; TJ14]. These setbacks can be caused by market changes, customer orders, company restructuring or similar unpredictable reasons.

Software developing companies are often faced with problems when estimating the effort needed to complete a software project; this is known as a major challenge for many software project managers. Due to the intangible nature of software, effort estimation is a crucial and critical activity in Software Engineering for planning and monitoring software project development. Under-estimation

can cause schedule and budget overruns as well as project cancellation. Over-estimation delays funding to other promising ideas and organisational competitiveness. Consequently, both over- and under-estimating would result in unfavorable impacts to the business competitiveness and project resource planning that can negatively affect the outcome of software projects [Men+17; Dej+11]. Having tools to track the project progress and the actual working hours would overcome these over time challenges and make timely and relevant decision-making.

Before going into the details of how an effort estimate is performed, it is useful to first define what effort estimation is. Some reviews provide evidence that the term 'effort estimate' is frequently used without sufficient clarification of its meaning [GJM06]. Launched by Brooks (1975) in his work "The Mythical Man Month"; its definition has undergone several connotations, but all are bound by the intent to search for the necessary resources needed to achieve project goals. Project Management Institute (2013) defines the estimation objective as providing an approximation (estimate) of the amount of the resources necessary to complete project activities and to deliver outputs (products or services) of specified functional and non-functional characteristics [TJ14].

Software effort estimation (SEE) is the process of predicting and measuring the most realistic amount of workforce and effort required to develop or maintain a software project, based on information collected in the early stage of a software project. Effort is usually expressed in units such as man-day, man-month and man-year; it defines the total time that members of a development team need to perform a given task or software product [Dej+11; Živ+11]. Precisely for this reason, SEE is an integral part of software project management. Even nowadays, research is focused on improving the effort estimation in software engineering.

1.1 The necessity of estimating effort

One might think that problems related to effort decision-making can be avoided simply by spending time and effort on a good initial design. There are different type of development processes, each having a different perspective on planning and estimation. During the past years the most widely adopted are the agile approaches [CH01], such as Scrum [SB02], leading to the formal use of the term Agile Software Development (ASD). Under an ASD process, software is developed incrementally in small iterations with fast and frequent changes to incorporate. These changes are influenced by customers' feedback, which servers as important input for subsequent iterations. This also implies that estimations and plans need to be done progressively. A tool capable of measuring these trends, as an independent activity, would provide more resources for future planning and help to understand whether the agile properties are respected or not. De facto also ASD is not exempt from the SEE process from which an active research area exists [Usm+14].

Software organisations do not regard estimation as a separate activity, but as an integrated part of project planning, project pricing and project budgeting. The reason is to prevent that software organisations propose unrealistic software costs, working within tight schedules and budget. It is a key factor to understand the cost involved in the realisation of a product, as well as a progressive study of software development process. It identifies the necessary project activities and how they are to be accomplished [Ker+22]. The benefits come from both sides: it helps teams to ensure a product is developed and delivered on time and it enables product owners to manage resources.

As aptly concluded by Barry Boehm (1981), "*Poor management can increase software costs more rapidly than any other factor, particularly on large projects*". Project managers should anticipate possible risks, evaluate their impact on effort, and adjust project plans appropriately. During estimation, project managers should consider the uncertainty of estimated values or the impact that known environmental factors may have on effort. The objective of risk response is to reduce its impact on project effort and thus to ensure that the project will

be finished within acceptable effort.

There are three major aspects to the project and its environment, which contribute to effort deficiencies: random changes, limited cognition, and limited granularity character. These three constraints lead us to the major types of effort estimation uncertainty: probabilistic, possibilistic, and granulation, respectively [TJ14].

This results in the evaluation of a new aspect of effort estimation. One should not expect effort estimation to ever be an exact science [Živ+11]. "*In the end, an estimate is just an estimate, it is not exact. After all, the process is called estimation, not exactimation*" - Phillip G. Armour. Accurate estimation is a complex process established in the preliminary phase between the client and the business enterprise. It can also be built upon the inspiration and generalisation from a small number of historical projects. So then it can be visualised as *software effort prediction* [BP10].

Depending on the weighed outcome of the effort study there is a different impact in the development steps. SEE manages the fate of the project approval, especially in the early stage of the project. A negative effort would lead to non-approval by the project manager who is responsible for planning and managing of the project [TJ14]. In order to perform efficiently, also the development team members must be aware of the outcome effort report from which they understand their individual roles as well as the overall activities of the team as a whole. Being able to measure the contribution of an individual developer is not an easy task. A good solution would be to combine the effort estimate employed by developers with an in-depth analysis of the tools and libraries used by them. This would also allow for more accurate profiling of the developer and his role in the team.

SEE is crucial at the early stage of project control activities. The staff resources or effort required for a software project are notoriously difficult to estimate in advance due to the minimal amount of information available. The later in the project runtime, the more we know about the project and the lower the likelihood of potential project changes. An obvious consequence is that the

earlier we estimate in the project, the larger the uncertainty we have to take into account due to less correct information on the actual project. Risks happen as result of insufficient information, which we can not know in advance. For this purpose, project managers and software development organisations have a great need for estimates at a very early stage in a project in order to appropriately tender for business and to properly manage resources [SSK96]. SEE purpose is to reduce uncertainty within the effort estimation process.

1.2 Effort drivers: what affects effort estimation

In principle, there are an almost unlimited number of largely unknown factors potentially influencing software development productivity and project effort. These may vary from the usage of tools, methods and technologies for supporting software project activities (development and management tool usage on project performance)[1] to the quality of information on which the estimate is based[2].

Among all existing effort drivers, it is not necessary to select a minimal set of factors in order to study the effort of a project. Indeed, the fewer factors we consider, the less overhead needs to be spent on collecting, analysing, and maintaining corresponding project data. On the other hand, the more relevant factors we consider, the better we are able to explain and estimate project effort [TJ14]. Usually a minimalist set of factors is typically dictated by limited resources for effort estimation.

Since software development is still a human-based activity, the main driver affecting the budget of project development, and therefore it is important on the effort study, are the number of employees, the capacity of employees and the

[1] Characteristics of software development processes are traditionally an important factor influencing development productivity and project effort. Knowing the impact of most relevant process characteristics on productivity provides the basis for reliable effort estimates and importantly also indicates potential process improvement opportunities.

[2] The goodness of estimates is directly dependent on input data analysed. Some input data are difficult to obtain, especially early in a program, and there is a risk of incomplete, unclear, inconsistent, ambiguous or even contradictory information. The data must be sorted.

time taken (schedule pressure). A shorter project duration requires more work to be done in parallel, which increases the chance that some products are based on incomplete or erroneous components (thus increasing the amount of rework). In light of this observation, proper planning of personnel effort is a key aspect for companies [Dej+11; BTB07]. If the performance of the assigned human resources differs from that assumed initially, or the resulting task duration is not acceptable, then the initial effort estimate needs to be revised, and the planning cycle must be repeated.

Simply putting a group of individuals to work together does not automatically guarantee organisational success. It's always wrong to believe that everything can be solved simply by increasing the number of employees working alongside a project. Even Brooks (1975), looking for the optimum number of people working on a project, rightly pointed out that time dependence, required for the completion of the system, upon personnel number is not linear [Živ+11]. The success or failure of a software project depends much on how good or bad are the team interaction, coordination, and communication. The number of communication increases exponentially with the number of involved team members. Larger teams require more overhead for work coordination and management but also increase the chance of miscommunication. For this reason, estimating effort is crucial because hiring more people than needed, in the hope of speeding up the development process, leads to a loss of income and hiring fewer people than needed leads to an extension of the schedule [BTB07].

The negative effect of increasing a team in order to shorten the project schedule is even more severe if we add people during the project runtime, especially when the project is already late. Such attempts to rescue late projects may in practice have an opposite effect. When introducing new staff into the project the performances suffer in two ways [TJ14]:

1. team members need to put in a certain effort to introduce new staff into project activities;

2. team members who are introducing new staff cannot work on the project activities during that time.

That is why the size and structure of the development team is a key success factors as project grows in size and complexity. The size of software is typically approximated in terms of the amount of functionality or structural size of artifacts delivered by the software engineering processes. As project scale increases in size and importance, it also grows in complexity as a result. The greater the system the greater the number of development team members as well as the greater the team, the greater the time required for mutual coordination of decisions [Živ+11]; having side effects in their cost, time-to-market, functionality, and quality requirements.

The number and complexity of project management activities, in addition to the unstable and poorly documented development process, are factors that influence the effort of a project. The more complex the software, the more mental effort it requires to analyse, understand and implement it. The best recommended strategy to overcome this problem is the classic "divide and conquer" approach [TJ14].

Other factors which impact on project performance and SEE process are the different types of existing project development environments. The human-intensive character of software development and rapid changes in software technologies make the software project a highly unstable environment. It is thus particularly important that a potential effort estimation approach not only provides information on relevant effort drivers but also explicitly considers critical characteristics of a project context [TJ14].

Software is required to support a wide variety of domains; it must always be faster, more intelligent, more dependable; it must require fewer hardware resources and be ever easier to maintain. Depending on the project context, several different factors entail software development productivity: the programming language used to implement software code, the application domain, the development type (new development, maintenance) and applied development life cycle.

The aforementioned context implications are categorical factors which have assigned labels instead of numerical values. Since their quantitative impact on

effort is difficult to determine, context factors typically represent categorical characteristics of a project environment. These aspects uniquely distinguish the development of a project and prevent the possibility of replicating the study effort used for one project on another not equivalent. The reasons are obvious as software development companies and their employees evolve with time: experienced employee could leave the company (bringing productivity back to the level it was before this employee was hired), new employees can be hired or lost, employees can become more experienced, new type of software projects can be accepted, the management strategy can be changed, new programming languages can be introduced, etc.

Monitoring some of these activities and aspects by means of suitable tools and metrics allows to have a more total control and management of the development team. In this thesis, new measures are provided to project managers to better weigh the capabilities of the development team, the time spent on the accomplishment of tasks and the consequent effort applied to achieve the aim.

1.3 Selecting Best Practices for Effort Estimation

The presence of so many effort driver influencing SEE was the main cause that dampened the starting idea to produce a unique, robust, accurate, predictive cost model. The complex nature of software development has been recognised since the earliest efforts to characterise and predict software costs, improve the planning of personnel and evaluate risk factors. Nowadays, it is a challenging objective to provide consistently the most accurate estimate model. Unfortunately, the diversity of software project, the non-stationary characteristics of the software development environment and dataset dependency mean no single model will likely produce the best results for all project types [WOM15; KKM13].

This led to a rapid change of views and plans by researchers. Over the last decades, a growing trend has been observed in using a variety of software effort

estimation models in diversified software development processes. This allows researchers to the ongoing publication of new, often complex, modelling approaches for effort estimation. The focus is always on estimating the software development costs, the required effort to develop a new project based on historical previous project data and preparing the schedules more quickly and easily in the anticipated environments. Being able to compare and determine the best effort predictor for different scenarios is critically important. Despite decades of research, there is still no consensus on which effort predictors are better or worse than others [KKM13]. What is certain is that large number of different techniques have been applied to the field of SEE, of fundamental importance became the possibility to choose the most appropriate software development effort predictor for the reference software projects. In this thesis, new metrics tools are provided to allow the study of the development team's contribution and support effort estimation, regardless the approach adopted by software managers.

Although it takes up to a decade and systematic literature reviews before a SEE become widely popular and accepted by the community, a great number of different effort estimations in the form of models, techniques, methods and tools have been developed. The proposed estimation models can be categorised, based on their basic formulation scheme, into model-based and expertise-based methods, but of no less importance are approaches based on analogy, size methods and combination of two or more of these models.

In order to determine how best to characterise the local environment, size metrics examine various artifacts produced by software project. Example project outputs include: software code, software executables, documentation, test cases, etc. The most common measure of software size is the program length quantified in terms of the lines of source code (LOC or SLOC) [TJ14]. A simple example of productivity deductible from the size metric is defined as effort in man-months divided by the number of code lines. Effort estimation models based on the number of code lines have one considerable shortcoming: the number of code lines is known only after the coding and testing, quite late in the lifecycle of

software development [Živ+11]. Notoriously LOC is a deprecated size metric to take snapshot of software complexity. However, it becomes useful when applied in the analysis of code evolution throughout the development of the software. An ad hoc metric was created in the following chapters.

Even so, there also exist other metrics of software size which can be calculated in the earlier phases of a project. The best-known and most widely used metrics among them is Function Points Analysis. Function points measure software size based on the functionality requested by and provided to the end user. FPA is applicable from early requirements specification phases and is independent of any particular programming language or technology used for implementation.

The effectiveness of size method comes in support of algorithmic and parametric models as input estimation variables. Examples of these models include COCOMO II (Constructive Cost Model) [CDB98] and schedule compression models such as SLIM [GS20]. Although the precise formulation of each model varies, they may all be considered as derivatives of the following form: effort= $\alpha * size^{\beta}$ where α and β are two factors that can be set depending on the developing company, productivity and economy of scale coefficient respectively.

All the effort models treated are commonly built and evaluated using a set of historical data. The usual approach involves separating the data into a training set (from which a model is built) and a testing set (from which the model's accuracy is assessed). Typically they exploit training data about past projects to build an estimation model which is then used to predict the effort for a new project. Such a model takes as input a set of predictors (e.g. manager experience or team experience) and returns a scalar value that represents the effort estimated to develop a new software system [SPH16]. Depending on the type of data collected, it is possible to construct a learning system that makes use of several machine learning methods on software effort estimation. There are many proposals in the literature including various kinds of regression (simple, partial least square, stepwise, regression trees), back-propagation and neural networks. The main reason for using such a learning system for this problem is to keep the estimation process up-to-date by incorporating latest project

data. If the effort estimation process evolves with a new project, the estimation model is kept up-to-date [BTB07]. The histograms obtained as results from the analysis metrics of this thesis could be normalised and used as vector values input data for new machine learning models.

Even parametric models are not free of drawbacks. These can suffer from the necessity of calibrating the model for each individual measurement before application in the concrete environment. Basha et al. [BP10] distinguish these technology factors that influence calibration in two macro sectors – technical and environmental. The technical aspects include those dealing with the basic development capability: organisation capabilities, experience of the developers, development practices and tools etc. The environmental aspects address the specific software target environment: CPU time constraints, system reliability, real-time operation, etc. The parametric downside models do not end here. The available models are: environments specific, subjective of input values and introduce rather complex weight coefficients (like with function points) [Živ+11].

When applying empirical parametric models, attention should be paid to make them as least complex as possible. The complexity increases with the level of detail of the model. A common modification among most of the models is to increase the number of input parameters and to assign appropriate values to them. Though some models have been inundated with more number of inputs and output features and thereby the complexity of the estimation schemes is increased, but also the accuracy of these models has shown little improvement [BP10]. E.g. COCOMO II used 31 parameters to predict effort and time [Boe+09] and this larger number of parameters resulted in having strong co-linearity and highly variable prediction accuracy, compared to other estimation schemes. Even if the underlying concepts and ideas are not publicly defined and the model has been provided as a black box to the users.

Several studies have assessed the applicability of data mining techniques to software effort estimation. The majority of models are based on automation techniques for data creation and gathering. However automating the process

necessarily involved making some assumptions, and the validity of results depends on those assumptions being reasonable. The data source is a meaningful case.

The accuracy of an estimate depends on the dataset characteristics. However, many studies evaluate modeling techniques on a particular, sometimes proprietary, data set which naturally constrains the interpretability of the observed results [Dej+11; KM09]. "*No one can be sure that the specific data sets were not selected because they are the ones that favor the new technique*" - Kitchenhamand Mendes. It is not always clear if the projects data studied are representative of the software industry, or even of the organisation from which they come. This is corroborated by the nature of the dataset: use of datasets difficult to obtain or are proprietary datasets and therefore not publically available [WOM15]. Because of this it remains possible that results cannot generalise beyond their data set. Our intent instead was to make available all the outcomes dataset collected in order to share the results and encourage study and research in this area. Researchers can take advantage of the effort data collected for their experiments, as well as replicate the same experiments. All estimation methods and their outputs can be reused partially or entirely in estimation contexts other than those for which it has been developed, because they are not bound by any case or limitation.

What is expected is that a similar positive behavior should be observed when applying real heterogeneous datasets from public domain or random sample of projects. Note that data sets from public domains come from different sources with various differences in project characteristic and evaluation criteria [KKM13]. Even though productivity varies significantly across projects and estimations can only be performed considering previously existing projects, Minku et al. [MY12] have investigated if cross-company data could help to increase performance and under what conditions. One of the reasons is that the number of projects available from a single-company is typically small, causing SEE models to perform poorly. If possible, it would be desirable to use cross-company data to improve the performance attained by models trained solely on

single-company data.

Traditionally, baseline productivity is typically determined based on experience gathered in projects successfully completed in a similar context. With comparable accuracy to algorithmic methods in some studies and potentially easier to understand and apply, an alternative popular approach to algorithmic models is the studying of effort estimation using analogy. The basis for estimation by analogy is to describe (in terms of variables or product value measure that acts as an effort driver) the project for which the estimate is to be made and then to use this description to find other similar finished projects. The productivity of similar already completed projects is employed as a basis for predicting the effort required to complete a new target project.

ANGEL [SSK96] is an example of automated analogy environment that support the collection, storage and identification of the most analogous projects in order to estimate the effort for a new one. Once the project reference data have been collected the analogies are found by measuring Euclidean distance in n-dimensional space where each dimension corresponds to a variable. It calculates the Euclidean distance between the target project and potential closest analogues.

Among the well-known factors that promote the use effort for analogy we can mention [WJ99]: it is easy to understand, it is useful where the domain is difficult to model, it can be used with partial knowledge of the target project, it has the potential to mitigate problems with calibration and outliers, it offers the chance to learn from past experience. A disadvantage of estimation by analogy is that it requires a considerable amount of computation. It requires the creation of an environment that supports the collection, storage and identification of the most analogous past projects [SSK96]. However it is also necessary to consider that an analogue may be selected and used regardless of its appropriateness. An old project could be selected as an analogue-one because it appears similar to the target project, although factors affecting effort have changed over time.

While effort estimation often requires generalising from a small number of historical projects, a factor to pay attention to is project age. Generalisation from

such limited experience is an inherently under constrained problem. Lokan and Mendes [LM09] have approached the dilemma: whether it is preferable use the entire historical data of past projects, or if it is more appropriate, in a rapidly process improvement and technological advances, to use a window of recent project. Particularly relevant is the chronological projects sequence, not casually. Specifically, if one assumes that recent projects better reflect current development tasks and practice, while past projects become less relevant over time, it might make sense to discard older projects.

Finding out which past projects can inspire a new one's effort is completely omitted from the expert judgment-base estimation model. Based on human expertise (possibly augmented with process guide-line, checklists, and data) to generate predictions, estimates are usually produced by domain experts based on their very own prior experience in software development. Due to its flexibility it can be applied in a variety of circumstances where other estimation techniques do not work (e.g. when there is a lack of historical data) [KKM13]. A number of different variations exist, e.g. Delphi expert estimation [Kit+02; RW01] in which several experienced developers formulate an independent estimate and the median of these estimates is used as the final effort estimation.

While still widely used in companies as domain strategy for effort estimation [MJ03], an expert-driven approach has the disadvantage of inherits all the weaknesses of methods based on human judgment [Dej+11; Uus+15], such as lacking an objective underpinning, subjectivity and dependency on the individual capabilities and preferences of involved human experts. No doubt the expert opinion is a non-algorithmic method. In fact the outcome of the estimate is not explicit and therefore not repeatable.

Each estimation method has its specific strength and limitations, depending on the particular environment and context in which they are to be applied. To go back to the original point, there is no single model suitable for every situation and environment. That is why, it is not wrong to adopt hybrid or multiple estimation methods approach. In these approaches, elements of different estimation paradigms, for example, expert-based and data-driven estimation, are

integrated within a hybrid estimation method.

The most important consequence of different combination estimation approaches is the potential to substantially improve the reliability of estimates [TJ14]. By using more than one technique it is possible to assess the degree of risk associated with widely divergent predictions, the possibility of validating alternative estimates against each other and investigate sources of potential discrepancies.

While hybrid should preferably represent different estimation paradigms and be based on alternative, yet complementary, information sources. When using multiple estimation methods, we must face the issue of applying multiple independent effort estimates to evaluate the same work. Their outcome estimates are combined to provide into a single final prediction. The key challenge is to reach consensus between multiple estimates.

A simple and most common way of handling estimation uncertainty is to provide a range of values, instead of a crisp estimate. A typical way of considering multiple estimates is to compute multiple uncertainty values, for each estimate individually and combine them either as an uncertainty distribution or as a single value synthesised using one of the common statistics over this distribution, for example, variance ranges, mean or median.

Trendowicz and Jeffery [TJ14] recommend using multiple estimation methods and combining alternative estimates only when: it is very important to avoid large estimation errors, the situation in which the estimation method is to be applied is uncertain (for example, the project environment has not been precisely specified yet, and estimation inputs and constraints are not clear), when there is more than one reasonable estimation method available to use different sources of information, or otherwise it is unclear which forecasting method is more accurate.

1.4 The proposed approach

This thesis presents several methods to detect and study new parameters useful to assist and validate effort and cost estimation in a software development

project. The results achieved have the purpose to improve development productivity, to justify project effort, to better understand the projects, the community behind them, and the behavior of employees; identifying, cataloging and creating comparison models between repository software to pave the way for future research in software engineering.

For this purpose, *data analysis* is of fundamental importance. Expression used to indicate a process of inspection, cleaning, transformation and modelling with the aim of discovering useful information, informing conclusions, and supporting decision-making [Bro14]. More specifically applied to the analysis of software repositories. The steps and processes that bring to this can be divided into various fields and selective environments. Online, not by chance, there are various specialised tools in this domain capable of operating independently and separately, offering for the most only visual tools and not methods or metrics for studying and analysing data to take part in possible future decision-making process and new software engineer applications. The metrics presented in this thesis succeed in this aim.

Mining Software Repositories (MSR) is a research field in which large amounts of data taken or available from a software repository are analysed in order to discover interesting and applicable information usable on software systems, projects and software engineering. It analyses the evolution of software systems in an automatic way, by applying data mining techniques in the history of the development data [Sun+16]. Studies in this area give the possibility to reveal important aspects of the development process for a project, or models in the evolution of the software, which could be generalised to other software systems [Bav16; JLW12].

While researchers increasingly recognise the potential benefit of extracting various types of information to support the maintenance of software systems and to improve the design, on the other side the presence of these large quantities of data in the repositories does not facilitate their direct and immediate analysis. Data may not be suitable for all types of research and data-misuse may lead to distorted results. They must be properly represented and processed to

obtain useful information through subsequent analysis for different scopes and purposes (MSR) [GS17].

A "software repository" is equipped with a source code control system and includes various information generated during the development of a software, such as: stored communications between project staff, bug reports and other aspects that surround a source code to help manage the progress of software projects. Up to now GitHub reports approximately more than 330 million repositories, supported by a community of about 83 million developers who learn, share and work together to develop largely high-quality software[3]. Certainly this implies a great possibility: detect and define design patterns that recur throughout the source code of the various projects. Actually it becomes increasingly likely that in these models are present knowledge of good development practices [McI+16]. The present work aims to make the reader know and understand the concept of software metrics and related aspects, analysing the possibility of applying these metrics in agile software development processes, in order to verify the presence of the various phases that make up the evolution of an Agile development.

These metrics can represent a standard of measure, quantification and evaluation of different aspects of software development. The advantages they offer are numerous; they can monitor the progress of the project (or product) software and ensure its quality. Indeed, it should be noted that the use of metrics in Agile processes are considered essential, as they allow to improve the predictions and management of software products.

The analysis of the relations between the metrics studied and implemented has been of fundamental importance in order to characterise the possible development processes and jobs used in the examined repository. This was done to find an effective correlation between the metrics used and the results obtained, and to validate the predetermined metrics.

Due to unformed view on how to assess estimation accuracy measurement and the lack of resources to the in-depth analysis of estimation accuracy data across project, Grimstad notes how most software organisations typically do not collect

[3]GitHub https://github.com/about

the data necessary to validate and adjust the actual effort to make it comparable with the estimated effort [GJM06]. For this reasons, this thesis aims to identify the characteristics and differences between various repositories in order to create patterns and structures to be used for the effort detect, diversified according to the repository application domain. These aspects are further deepened in the development and creation of individual repositories, also in relation to expose each different weight contribution in the project implementation.

This leads us to consider the possibility of using effort estimation progressively throughout the development process. As the project work progresses, the project manager should so compare estimates against actual project values and clarifies potential sources of deviations. Based on the revised information regarding the actual project environment and scope, re-estimation takes place in order to account for potential discrepancies from the plans and changes to the project environment and scope.

With regards to the different aspects of the data analysis and extraction from repositories, which will be treated in detail later, from my work contribution it is possible to distinguish between: monitor team performance, individual contribution effort versus team, the presence of regular steps in the code development that must be respected and a vision of the learning skills of their employees.

1.5 Thesis structure

In Chapter 2 all the required fundamentals for the works described in further chapters will be discussed. The chapter is composed of different sections, starting from the theory of *agile practices*. In the context of this description, the fundamental principles and objectives of this system have been examined, including the practical implications for experimentation. In this context, it is useful to describe the *data mining* as a field of analysis of the development of repositories, which allows to discover useful information related to large data repositories. Moreover, concept related to *git hosting service* gives the fundamentals which were of great importance for the works presented in this thesis.

In Chapter 3 a suite of process metrics related to the development process will be proposed, which reveal the effort of developers and their practices analysing data extracted from repositories. The metrics are able to determine the level of workload constancy of developers but at the same time can act as a repository classifier: Open Source, Side Project and Full Time Project.

In Chapter 4 the agile Scrum framework will be deepened and with it will be formulated and implemented a couple of metrics to offer companies the opportunity to extract workflow behaviours from the development progress undertaken by Scrum team. The main purpose will be based on automatism Sprint reveal, trying to differentiate the phases of testing and development among all branches that make up the project repository.

Chapter 5 focuses on the study of estimation measurement systems on API misuse effort. In finding what concerns the effective contribution of members on a development team, a series of metrics have been implemented to measure effort by different degrees levels of APIs use. Through a detailed analysis of all APIs used in the history of the project and how correctly they have been applied, it's possible to profile actual roles and contributors of repository depending on the efforts that have been applied to the repository.

Finally, Chapter 6 gives the conclusion for this thesis, while Appendix A gives two code listings that are referred by Chapter 4 to show the application of the PyDriller framework and its combination with a Bag of Words model on commit message.

1.6 Published Papers

Part of the work presented in this thesis is based on these co-authored papers:

- L. Pelonero A. Fornaia, E. Tramontana, *"From Smart City to Smart Citizen: rewarding waste recycle by designing a data-centric IoT based garbage collection service"*, in: Proceedings of IEEE International Conference on Smart Computing, SMARTCOMP, Bologna, Italy, 2020.

- L. Pelonero A. Fornaia, E. Tramontana, *"A Blockchain handling Data in a Waste Recycling Scenario and Fostering Participation"*, in: The IEEE Second International Conference on Blockchain Computing and Applications, BCCA, ANTALYA, TURKEY, 2020.

- Camuto Enzo, Fornaia Andrea, Pelonero Leonardo, Tramontana Emiliano *A Suite of Process Metrics to Capture the Effort of Developers*, in: ACM 10th International Conference on Software and Computer Applications, ICSCA, Malaysia, 2021.

- Pelonero Leonardo, Tramontana Emiliano *Comparative analysis of software repository metrics to estimate developer contribution* (draft)

- Pelonero Leonardo, Tramontana Emiliano *Discover Scrum Practices form Repositories to Suggest Improvements* (draft)

- Pelonero Leonardo, Tramontana Emiliano *How to measure effort misuse on APIs call* (draft)

Chapter 2

Background information and theory

In this chapter all the required fundamentals for the works described in further chapters will be discussed. This section describes what it means to be Agile, provide a definition of the agile model, the principles and methodologies for adopting agile software development. While Agile techniques vary in practices and emphasis, they share common characteristics, including iterative development and a focus on interaction, communication, and the reduction of resource intensive intermediate artifacts. The importance and usefulness of Mining Software Repository field and the processes that characterise it analysing the rich data available in software repositories. The role of one of the most used tools by software developers Git, the terminologies that concern it and the commit structure.

2.1 Agile software development model

The field of software development is not shy of introducing new methodologies. Indeed, in the last 25 years, a large number of different approaches to software development have been introduced, of which only few have survived to be used today. Agile development refers to a set of software development methods that have emerged since the early 2000s and are based on a set of common principles,

derived directly or indirectly from the principles of "Manifesto for Agile Software Development"[1].

The notoriety of agile methods, it has generated a lot of interest among practitioners, derives from the fact that they are opposed to traditional models. They propose a less structured approach but focused on the goal of delivering to the customer working and quality software in early and frequent delivery.

The practices promoted by agile methods allow the formation of small, polyfunctional and self-organised development teams, as well as iterative enhancement and incremental development, adaptive planning and direct and continuous customer involvement in the development process.

Most agile methods are aimed at reducing the risk of failure or encountering development issues. To prevent such possibilities, it has been decided to develop software in limited time windows, called iterations that usually last a few weeks. Each iteration is a small project in its own and must contain everything that is needed to release a small increase in software functionality: planning, requirements analysis, design, implementation, testing and documentation.

Developing in iterations allows the development team to adapt quickly to changing requirements. Even under the hypothesis that the outcome of each iteration does not have enough functionality to be considered complete, it must be published and, in the next iterations, the established demands of the customer must be remedied and satisfied. At the end of each iteration, regardless style of development, the team must re-evaluate project priorities.

Agile methods are mainly based on real-time communication, preferably face-to-face. The agile team, in fact, consists of all the people who know the goal and the customer's request. Reducing intermediate artifacts that do not add value to the final deliverable means more resources can be devoted to the development of the product itself and it can be completed sooner.

The developer team and their customers can also debate during the software development phase, this aspect is certainly positive from developers' point of

[1]Beck, K., et al. (2001) The Agile Manifesto. Agile Alliance. http://agilemanifesto.org/

view in order to allow them to better understand customers' requirements, negative for unexpected last minute requests [BA04].

It is interesting to note that there is a lack of literature describing projects where Agile Methods failed to produce good results. Instead, there are many studies reporting poor projects due to a negligent implementation of an Agile method [CLC03]. This leads to the concept of analysing for which situations agile methods are more suitable than others; there is no one-size-fits-all software development model that suits all imaginable purposes. As stated by Hawrysh and Ruprecht (2000) it is up to project management identify the specific nature of the project and accordingly select the best applicable development methodology [Abr+17].

2.1.1 Principles and Objective

The aim of agile development is not only the fulfilment of the contract, but also the complete customer satisfaction. The correct use of these methodologies, moreover, can reduce the costs and the time of development of the software, increasing its quality.

The principles and focal values on which an agile methodology is based following Agile Alliance are [Amb12; BA04]:

- Individuals and interaction over process and tools: people interactions and close team relationships are more important than processes and development tools. The agile movement emphasises the relationship of software developers and the human role reflected in the contracts (i.e. the relationships and communication between the actors of a software project are the best resource of the project);
- Working software over comprehensive documentation: new releases are produced at frequent intervals, and the code must be kept simple and technically advanced as possible, thus reducing to minimum documentation;

- Customer collaboration over contract negotiation: collaborate with customers as well as respect the contract (direct collaboration offers better results than contractual relationships). The negotiation process itself should be seen as a means of achieving and maintaining a viable relationship;
- Responding to change over following a plan: be ready to respond to changes emerging during the development process life-cycle (therefore project stakeholders should be ready, at all times, to change the priorities of work in compliance with the final goal).

As mentioned, agile methodologies allow continuous review specifications, adapting software development progresses, through an iterative and incremental framework and a strong exchange of information and opinions between the developers and client.

2.1.2 Practices and Methodologies

The individual practices applicable within an agile methodology are numerous and depend essentially on the needs of the company and the approach of the project manager. What must take into account are the characteristics of each practice, the benefits it brings and the consequences it entails. For example, in Extreme Programming, the absolute lack of any form of design and documentation is compensated by the strict involvement of the customer in development and pairs programming.

The most widely available practices are similar and can be grouped into the following categories [BA04]:

Automation - Agile methodologies focus on programming without engaging in side activities, these latter can be eliminated or automated. For example, delete documentation by increasing testing, but you can't delete both; so you have to choose the path you want to take and make sure to use tools to automate as many side tasks as possible;

Customer involvement - There are different degrees of possible involvement; for example in Extreme Programming the involvement is total, the customer even

participates in weekly meetings of programmers; in other cases, the customer is involved only at the first design step; in others the customer still participates indirectly and is used as a tester of the released version;

Close communication - This practice is perhaps the only true nodal aspect that makes a methodology agile. Close communication, in fact, means interpersonal communication, between all the actors of the project, including the customer. This serves to obtain a good analysis of the requirements and a profitable collaboration between programmers even in an area of almost total absence of documentation;

Frequent deliveries - Making frequent intermediate versions releases of the software give multiple advantages at once: can start the iteration already having a block of working code; offers the customer something to work with and distracts him from any delivery delays of complete project; the customer attends as tester, note that he will use the software and will detect any anomalies; more precise information is obtained from the customer requirements that he would probably not have been able to express before;

Hierarchy - The choice to create a hierarchical structure within the development team depends greatly on the project manager's approach. Indeed, this choice comes with compromises: with a hierarchical tree structure you get the opportunity to manage a very high number of programmers and work on different aspects of the project in parallel; if you decide for a total absence of hierarchy you will have a very compact and motivated development team, but it necessarily small in terms of number of programmers;

Iterative development - an important practice through which a starting "idea" (a concept, a proposal, a set of needs) evolves to become a product of value for customer. Iterative enhancement works through cycles of activities that do not change, but that repeating periodically lead to the 'raw' solution, that refine to become the final product;

Pair programming - development is done by pairs of programmers;

Refactoring - the restructuring of parts of the code that keeps its appearance and external behavior unchanged;

Reverse engineering - automatic generation of documentation from the code already implemented. Popular practice but not very exploited. It is a time saving practice but often the produced documentation is unusable, or is produced only for a bureaucratic request of the customer and will never be used really;

Test Driven Development - testing to be performed during the project. Only after the actual tests have been passed, it is possible to continue in the following implementation steps.

Versioning - as consequence of iteration in production a tool to control the versions of the software produced and released is introduced. One of the most widely used and suggested tools is CVS.

2.2 Mining software repositories to assist developers and support managers

The amount and data size present in software projects is constantly increasing. As a result, this complicates the work of developers and maintainers. In recent years, researchers analyse software repositories to better understand their ongoing structural change due to the increasingly established presence of long-term projects.

Thomas Zimmermann, describing one of the main goals of MSR (Mining Software Repository) states that "Learning from past successes and failures helps to create better software" [Zim06; WH05b]. However, learning from history is not an easy process as the software evolves over time. Understanding the software evolution process is a difficult task. Major systems are used to have a long development history, with numerous different developers working on different parts of the system. It is natural that no developer knows the entire source code of the project. For this reason, the idea of a manual analysis on all types of software is impractical.

Most of software projects cost is related to the reuse of components or the maintenance of legacy systems software (outdated or obsolete software). It follows that the study and knowledge of past projects patterns are very useful to understand. The MSR field becomes a key to support maintenance, improve software process quality, and empirically validate various research ideas or techniques.

MSR is described as "A field that analyses the rich data available in software repositories to uncover interesting and actionable information about software systems and projects"[2].

The definition of MSR is similar to Data Mining, which is defined as the process by which automatically discover useful information in large data repositories [TS05]. In fact, Data Mining is a more general field of MSR. Data mining consists in discovering interesting and possibly unexpected patterns in a data system. Most data mining analysis is based on discrete data, nominal or textual associated to business concepts. These informations can be used to infer new knowledge, as a decision support tool or as a basis for other operations (e.g. user profiling). There are two secondary tasks associated with this primary task: data cleaning, pre-processing detection and cleaning of artifacts data; data display, to show the results of the data mining conducted.

Error and effort predictions are important tasks to reduce costs in a software project. The research activity and effort identification on a software project aims to understand the necessary path to complete the started project. Predicting an error fault, however, helps the detection of modules subject usually to bugs and errors, so that we can avoid them and draw the necessary and useful considerations. Such forecasting and monitoring models should not only be accurate but also understandable to the end user, who can then exploit the benefits on later business decisions [Moe+15].

Among the most common areas in MSR we can cite [WH05b]: software evolution, models for the software development process, characterisation of developers, forecasting software quality, detecting software bugs, analysis of changes in work patterns and detection of duplicate code.

[2]International Conference on Mining Software Repositories http://www.msrconf.org/

The reason behind the growing need of MSR is due to the increased amount of unstructured data available, the use of Issue tracking system, as well as the relevant communications and interactions that developers undertake in the implementation of the project. The extraction of such data represents an unprecedented opportunity for researchers who want to investigate, ask new research questions and build possible maintenance systems that support the various development activities. Such applications thus enrich software engineering beyond the difficulties of unstructured data study [Bav16].

MSR supports researchers in this field, who try to achieve the criteria to acquire features related to the evolution and changes of the software, following the relevant steps, such as: the calculation of metrics, the extraction of data and the recording of statistical developments. It analyses the evolution of the software automatically, through the application of Data Mining techniques in the history of data development.

Primarily MSR begins with the extraction of interesting data from various large repositories such as: source code control systems, bug tracking systems or communications archives and so on. After data extraction, starts a filtering and conversion phase, the data is converted in appropriate data structures [JLW12].

- Data extraction: where the raw data were extracted from;
- Processing: what type of data were managed;
- Analysis: which algorithms are used to analyse data;
- Evaluation: how MSR outcomes were evaluated.

An important aspect of mining software repository is the analysis of the source code itself. There are several billion Open Source lines of code online, and a large part of them have professional quality. In practice, it entails a great possibility: finding and defining models that recur throughout the source code of various projects. Studies in this area can reveal important aspects of the development process for a project [McI+16].

Mining could enable a wide variety of software engineering tools, e.g. to recommend which classes are most reusable [Era+19; RI19; GRF17], to understand how to reach and obtain highly configurable systems according to the needs that can be encountered in time [DDP18], or for identifying potential bugs, etc. according to statistical analysis results [Rad+13; DLR10].

Analyses are usually refined during several iterations through a cyclic data selection, pre-processing, and the construction of appropriate models and their validation. These study of software evolution activities are time consuming and errors prone. Best results are usually achieved by combining models of different techniques, which require a wide variety of tools integrated within the system.

There is no doubt that MSR studies benefit from automation as data is too large to be analysed manually, especially if you need to access data from many sources and combine them to get more comprehensive results and analysis. As a result of this, the choice falls on the use of MSR framework that supports the following advantages: (i) the researcher can focus on the objectives and not on the infrastructures, moving all the interest only to the possible results to obtain; (ii) the encoding of a framework improves standardisation and reproducibility of the experiments on the repositories (the possibility of make the raw data and the research code available so that others can achieve the same results as shown on conclusions of the research work).

Researchers continue to demonstrate the benefits of the Mining Software Repository on software development activities. However, because the mining process takes a lot of time and resources, they often rely on distributed platforms and parallel programming optimisations to accelerate and expand their analysis. These adopted platforms are specific to the framework and the type of analysis to be performed, difficult to reuse in other contexts and further research, and offer minimal support for debugging and deployment [Sha+09].

2.2.1 Data extraction

Data is one of several important aspects of software-based systems. Most, if not all, applications are based on moving, utilising, or otherwise manipulating some

kind of data, after all. The major difficulty in collecting the data is the scarcity of data due to the unwillingness or lack of data collection at the companies' side. For this reason we have used data from public data repositories and we also collected data from open source software foundation (CNCF).

The software is flexible and can deal with various data sets, drawn from a range of different environments, both in terms of the number of observations (projects) and in the variables collected. There are several data sets made available by research entities in the field, each particular for its properties and context of use. But a data sets typically contain a unique set of attributes that can be categorised as follows:

- Size attributes are attributes that contain information concerning the size of the software project. This information can be provided as Lines Of Code (LOC), Function Points, or some other measure. Size related variables are often considered to be important attributes to estimate effort.
- Environment information contains background information regarding the development team, the company, the project itself (e.g., the number of developers involved and their experience), and the sector of the developing company.
- Project data consist of attributes that relate to the specific purpose of the project and the project type. Also attributes concerning specific project requirements are placed in this category.
- Development related variables contain information about managerial aspects and/or technical aspects of the developed software projects, such as the programming language or type of database system that was used during development.

Data sets can range from larger and more heterogeneous to data sets that are smaller and more homogeneous, working in progress project or involve fewer completed projects per year.

In-depth studies and triangulation may be needed to ensure that all the data are based on the same project conducting surveys or logging estimation information.

To develop a study in this field, it seems necessary, in a first phase, to collect large amounts of data, often even from different projects, and store them on appropriate servers or workstations.

2.3 GitHub

GitHub is a hosting service for software projects and version control, in order to meet project and organisational needs to work and share the code development for all those who work on it. It commonly provides free services to Open Source developers, including project hosting, version control, bug and issue tracking, project management, backups and archives, along with communication and collaboration resources.

It is nothing more than an evolution of local and centralised Version Control System (VCS), which allowed programmers to maintain a database of versions and changes made on one or more files in the machine itself. What was done in the machine was the check-out, a synchronisation copy between the version currently working on (in Git work directory) with a specific version within the local database. It was thus possible to keep everything in an orderly manner. The only problem was that it was a purely local system; it was not possible to collaborate on a team on the same project [Spi17].

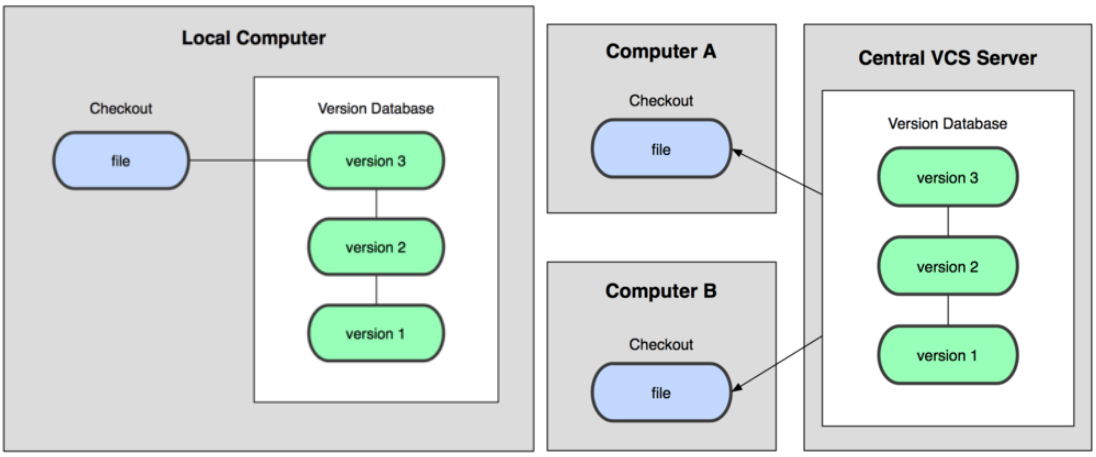

FIGURE 2.1: Comparison: Local VCS and Centralised VCS

Unlike local VCS and centralised VCS (figure 2.1), Github is a distributed VCS (figure 2.2). A fully mirror system that allows you to retrieve the entire code history from every single actor that have worked there (even in the unfortunate case of server malfunction). Giving great strengths and robustness to the system.

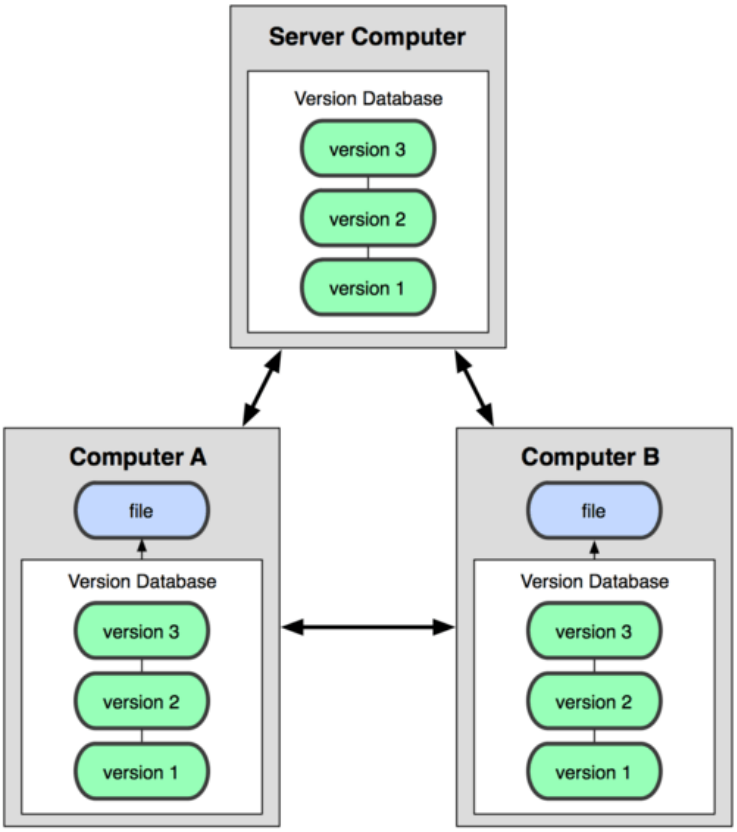

FIGURE 2.2: Distributed VCS

The local and the remote approach are combined, allowing to manage a large number of workflows, different ways to coordinate the developers participating in the common project.

If on one hand it becomes more difficult have general point of view of the project, on the other hand distributed VCS avoids problems of scaling for very large projects as well as problems of *Single point of failure*, on which if that single reference database on the remote server falls you lose the whole project and code history [Spi17].

2.3.1 Why to use GIT

This system is used primarily for its simple design that revolves around the commit graph and also because it is nothing but cumbersome or tedious.

It has strong support for non-linear development, which is one of the biggest differences from SVN where everyone planned the same development line to follow and respect. This allowed a single evolution version of the code over time. It is not like that in Git, since there is the possibility of creating branches on branches of code versions.

This drastic change development from linear to non-linear development is due to the fact that any branching operation has become an economic operation and simple to manage. Git becomes fully distributed and efficiently for both in terms of speed and for managing large projects[3].

Among other benefits, Git allows to do offline operations. Every developer has a local database and this allows to work remotely, without the need to be assiduously connected to create a saved code just modified. Indirectly this makes it possible to secure a possible change that has been done in strait schedule or that has not yet been fully review, avoiding saving it as the only solution for the whole team of work from which they will then take reference to go forward. These local changes will be considered only when you are aware that you can publish them at all, and not before.

2.3.2 Terminology

Commits are the core building block units of a Git project timeline. A Commit is nothing more than a snapshot of the state of a project, including files of various extensions which it was decided to have Git monitoring.

The copy of the project of each user is a snapshot and around it there are a series of additional information, as shown in figure 2.3, that are:

- commit author: who made that specific commit, who saved that code state;

[3]GitHub https://github.com/about

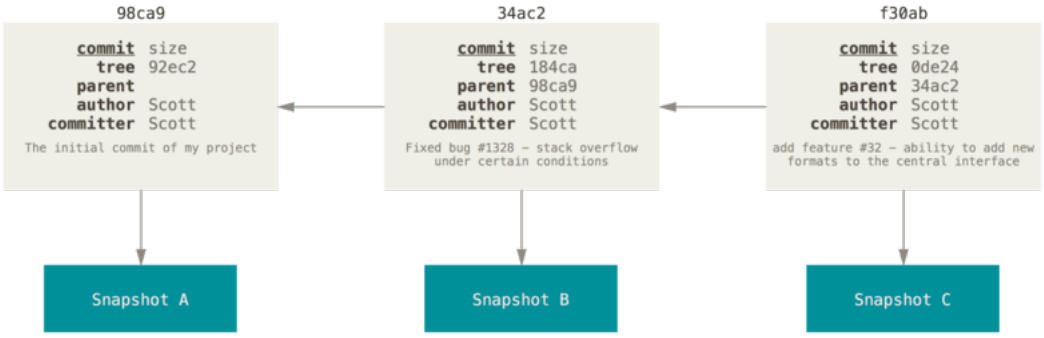

FIGURE 2.3: Commit sequences

- committer: who pushed the commit to the repository, which in some cases may also be different from its author;
- hash: hash that uniquely identifies the commit;
- message: a textual information put by developer to describe the changes made in the commit;
- list of modified files in the commit;
- list of commit parents: represent a reference to previous commits from which the current commit derives: 0) initial commit of project, 1) a single previous commit, or 2) if commit comes from the merge of two previous commits.

There are three types of pointers within the Commit Graph 2.4: Branch, Tag and Head. Head is a pointer that refers to the currently commit that is on the working directory, keeping a reference to the project version currently working on. Branch represents an independent line of development, it offers a way to work on a new feature without affecting the main codebase. Once the feature development is done, the branch will merge to the main commit to integrate officially the new feature. As well as, the utility of a branch can be to solve a specific problem that may require more commits but still refers to a specific problem (Hotfix). While Branches are pointers that can be moved, Tags are pointers that never move. They associate a label to a specific commit, used to annotate versions of the code.

FIGURE 2.4: An example of Commit Graph

2.4 PyDriller

PyDriller [SAB18] is a Python framework that helps developers on mining software repositories. It is capable of simplifying tasks, analysing a Git repository, allowing researchers and professionals to focus on their research and not on the manipulation details of Git itself.

It can easily extract information from any Git repository, such as commits, developers, modifications, diffs, branches and source codes, and quickly export csv files. It is a flexible MSR tool, the framework has the ability to perform arbitrarily analyses according to objective requirements.

Compared to many other tools that require specific structures and studies for their usage, PyDriller stands out for its minimalist structure and required APIs. It only requires storage and computational capabilities when necessary; there is no significant pre-processing phase or requests for large data bases. In this way, users and researchers are offered only the features to perform the required MSR tasks, hiding the structural complexity to end user.

The PyDriller architecture simply requires the repository path to be analysed or the Git url repository which will be directly downloaded; so that once, based on that path, the framework will return the repository commit list.

While MSR plays an important role in software engineering research, few tools

have been created and made public to support developers in extracting information from Git repository. However, the operation of extracting from repositories is not trivial. There are different types of frameworks and libraries in various languages that use the REST API calls (e.g. Github). These tools, however, are often difficult to use. One of the main reasons for this difficulty is that they incorporate all the features of API query, so developers are forced to write long and complex implementations even to extract a single data from a Git repository.

Depending on the level of authorisation, an API request allows application to request high level data or services such as creating repositories, modify or delete personal existing repositories. This can be achieved by using private credentials or by enabling access restrictions with special tokens created for the application (OAuth 2.0)[4].

By means of access tokens it is possible to take advantage of two great feature: revocable access, on which users can revoke permissions to third-party application and limited access, users can verify the access and permissions provided by a token, before authorising a third-party application, as well as control the traffic rate limits API requests. The number of possible calls requests is low that it requires desired delay time to avoid the total rate consumption offered and to keep the session running until the end of data extraction and and its consequent analysis. This system is used to prevent any abuse of services or principle of denial-of-service attack.

Normally tokens are created through web interaction. The application sends users to access on GitHub. GitHub thus presents a dialog indicating the name of the app, as well as the authorisation levels that the app will have once it is authorised by the user. At the end, GitHub redirects the user back to the application.

PyDriller does not manage or require any credential or access password, thus avoiding all the processing required by OAuth 2 and preventing from forgetting pendants and working tokens. The framework takes as input the repository,

[4]GitHub Docs https://docs.github.com/en

it will make a *clone* on a temporary folder, so that once finished the analysis phase will be deleted.

2.5 Open Source Software

Open Source Software (OSS) paradigm, can also be discussed from a philosophical perspective. It suggests the source code to be freely available for modifications and redistribution without any charges, in order to empower future innovation. This development process inspires a novel software development paradigm and offer an innovative way to produce applications taking advantage of global developer collaboration [Dem+02].

Feller and Fitzgerald [FF00] present the following motivations and drivers for OSS development:

1. Technological; the need for robust code, faster development cycles, higher standards of quality, reliability and stability, and more open standards and platforms
2. Economical; the corporate need for shared cost and shared risk
3. Socio-political; scratching a developer's "personal itch", peer reputation, desire for "meaningful" work, and community oriented idealism.

Most known OSS development projects are focused on development tools or other platforms that are used by professionals who have often participated in the development effort themselves, thus having the role of the customer and that of the developer at the same time.

Unlike ASD, OSS is not a suite of well defined and published software development methodologies and practices. Instead, it is better described in terms of different licenses for software distribution. However, it must have some structure, or else it would never have been able to achieve such remarkable results as it has in the past years.

Even though Cockburn (2002a) notes that OSS development differs for its own peculiarities from the agile development mode in philosophical, economical, and

team structural aspects, OSS does in many ways follow the same lines of thought and practices as other agile methods.

For example, the OSS development process starts with early and frequent releases, and it lacks many of the traditional mechanisms used for coordinating software development with plans, system level designs, schedules and defined processes.

Table 2.1 shows how the Open Source Software (OSS) paradigm places itself between the agile and plan-driven methods. The OSS is still fairly new in business environment and a number of interesting research questions remain to be analysed and answered. Thus the OSS approach can be seen as one variant of the multifaceted agile methods [Abr+17].

Even though it is possible to depict the OSS software development methods with the above iteration stages, the interest lies in how this process is managed, as can be seen how the OSS development method is characterised by Mockus et al. (2000) with the following statements:

1. The systems are built by potentially large numbers of volunteers
2. Work is not assigned; people themselves choose the task they are interested in
3. There exists no explicit system level design, or even detailed design
4. There is no project plan, schedule or list of deliverables
5. The system is augmented with small increments
6. Programs are tested frequently

Sharma et al. (2002) state that OSS development projects are usually divided by the main architects or designers into smaller and more easily manageable tasks, which are further handled by individuals or groups. Volunteer developers are divided in individual or small groups. They select freely the tasks they wish to accomplish. Thus the rational modular division of the overall project is essential to enable a successful outcome of the development process. Furthermore, these sub-tasks must be interesting to attract developers. Even though hundreds of

volunteers may be participating in the OSS development projects, usually there is only a small group of developers performing the main part of the work.

To work successfully, the geographically dispersed individuals as well as small groups of developers must have well functioning and open communication channels between each other, especially as the developers do not usually meet face-to-face.

To start or to acquire an ownership of an OSS project can be done in several ways: to find a new one, to have it handed over by the former owner, or to voluntarily take over an ongoing dying project (Bergquist and Ljungberg 2001).

The OSS development process can be seen as a massive parallel development and debugging effort.

TABLE 2.1: Home ground for agile and plan-driven methods (Boehm 2002), augmented with open source software column.

Home-ground area	**Agile methods**	**Open source software**	**Plan-driven methods**
Developers	Agile, knowledgeable, collocated, and collaborative	Geographically distributed, collaborative, knowledgeable and agile teams	Plan-oriented; adequate skills; access to external knowledge
Customers	Dedicated, knowledgeable, collocated, collaborative, representative, and empowered	Dedicated, knowledgeable, collaborative, and empowered	Access to knowledgeable, collaborative, representative, and empowered customers
Requirements	Largely emergent; rapid change	Largely emergent; rapid change, commonly owned, continually evolving – "never" finalised	Knowable early; largely stable
Architecture	Designed for current requirements	Open, designed for current requirements	Designed for current and foreseeable requirements
Refactoring	Inexpensive	Inexpensive	Expensive
Size	Smaller teams and products	Larger dispersed teams and smaller products	Larger teams and products
Primary objective	Rapid value	Challenging problem	High assurance

2.6 Conclusion

In this chapter all the fundamentals and background, to understand and evaluate the contributions presented in further chapters, have been presented. A great importance has been given to concepts related to data mining, theoretical description of the foundations of modern software development, identifying the characteristics of the Agile software development model to introduce the agile framework Scrum since needed to support the work presented in the following Chapters.

Chapter 3

Code History Metrics to classify Software Repositories at scale

Software repositories are a valuable support for large teams working together on a project. Besides the code, a repository includes useful data related to development practices. Being able to extrapolate these information would allow to understand and realise the typology of working procedures adopted by the team. In this chapter, a calibrated analysis approach will be proposed; a suite of process metrics is applied to data extracted from repositories and related to the development process. Complete automation has been achieved for extracting data from repositories and for computing metrics, as well as representing them. Only by having an overall view of the development method used and the productivity employed you can have total control of the project progression. The value of the proposed metrics is to reveal the effort of developers and their practices, in order to highlight strengths and weaknesses, then suggest improvements.

3.1 A suite of Process Metrics to Capture the Effort of Developers

Over the years, interest has increased how software developers approach their jobs and in the same way emulate its positive characteristics. However, it is not enough. We need to understand and be sure if developers benefit from

these techniques, tailor them, and apply them appropriately within specific environment. The aim of this thesis is the study and identification of metrics that define the work behind software project and describe what techniques the developers benefit from.

An important source of study are software repositories. It contains historical and valuable information about the overall development of software systems. Mining software repositories (MSR) is nowadays considered one of the most interesting growing fields within software engineering. MSR focuses on extracting and analysing data available in software repositories to uncover interesting, useful, and actionable information about the system.

One of the bases to foster collaboration and work in large development teams is the use of appropriate tools to converge ideas and communications. Hence, the need to avoid losing the main goal of a project. Software repository is one of the well known tools. For this reason, beside the code, a repository includes useful data related to development practices.

The term *software repository* includes all the aspects created during the software development such as: source code's control systems, archived communications between the project team, bug reports and other aspects that surround a source code to manage the progress status of software projects (Git repositories hosted by GitHub, Bitbucket, GitLab).

During development, dictated by schedule and time consuming tasks, new features and other code changes are implemented by developers. Software developers publish their source code in order to foster continued innovation in computing. Such code changes (or commits) must submit to rigorous activities of continuous integration tests and code review prior to merge into the main branch [GPD14].

In fact, it is increasingly likely that knowledge of good software engineering practice is hidden in these models. The software engineering practices must be an integral part throughout the life cycle (from planning, development stage,

to release preparation), so that teams can follow the best practices to prevent software defects and become more and more expert programmers [Rol+18].

Professionals and researchers are recognising the benefits of extracting these valuable historical information: to support software system maintenance, to improve software design and eventual reuse of it, to validate new ideas and techniques.

The use of process metrics on the development history is increasingly taking hold parallel to code metrics. For a long time, researchers have been interested in which classes of metrics (process or code) are better for defect prediction. According to Foyzur Rahman et al., researchers mostly focus on two classes of metrics: *code metrics*, which measure properties of the code (e.g. size and complexity), and *process metrics* (e.g. number of changes, number of developers) [RD13; MJ15], suggesting the use of process metrics as most effective for defect prediction.

Development practices have been at the center of previous studies and investigations. Catolino et al.'s survey [Cat+19] emphasises how many researchers found significant correlations between developers' experience and testing effectiveness, related to software maintenance and testing [BR08]. Evidence of this is provided by Pham et al. [Pha+14] who come to the conclusion that junior developers do not see the need to write test cases, as they have likely not experienced the consequences of inadequate testing.

In this chapter we propose to extract knowledge from a software repository by devising a set of metrics for a project. The proposed metrics are: Commits per Day of the Week (CDW), Commits per Hour of the Day (CHD), Average Commit Distribution (ACD), Commits per Week in the last Year (CWY), Changes per Week Trend (CWT), Lines Of Code in Time (LOCT).

Such metrics allow us to identify characteristics and differences in software projects, have an appropriate recognition profiling of the repositories, and consequently the developers behind them. Further outcomes are: the study of commits and trends of a developer, defining the correlation between developers'

experience and the effectiveness of activities and effort. Establishing the developer's workflow and his resulting skills is one of the qualities sought within a company in order to understand and determine the contribution of each individual.

The rest of the chapter is organised as follows. Section 3.2 describes the related work. Section 3.3 introduces our approach. Section 3.4 describes the implementation adopted. Section 3.5 presents the proposed metrics. Finally, Section 3.6 contains the conclusion.

3.2 Related Work

One of the advantages of MSR is the ability to use process mining techniques to extract knowledge from a variety of information sources. The study that leads to the knowledge of programmer behaviour try to find resources from all sides. Exactly as we did, in [CA16] authors tried to find ideas outside the mere source code management system and issue tracking tools. Due to the absence of a study on the direct developing process itself and in parallel to increasingly establishing of cloud-based IDE (last one GitHub's Codespaces[1]) they build up a plugin for Eclipse IDE to collect data in JSON to understand the developer process pattern and characterise developer roles along several machine learning approaches.

MSR is not only a dedicated practice by experts and researchers in the field, even companies invest a lot in this area. The organisations are looking forward to improve their software development processes to employ agile best practices. These practice are difficult for organisations to validate (e.g. customer collaboration could not be detected in the event logs).

Due to the increasing importance of open source software, modern software development processes involve multiple developers and development teams residing across different continents and time-zones [PSV11]. Such projects are

[1] GitHub Codespaces https://github.com/features/codespaces

usually managed by small number of developers, frequently working as volunteers, for these reasons there is often the risk to became unmaintained projects. In [Coe+18] authors proposed an approach to identify GitHub projects that are not actively maintained. Ten trained machine learning models were tested to identify the project status, based on a set of features about project activity (number of commits, forks, issues, and pull requests). The status includes: finished projects, deprecated projects and stalled projects. Despite the results obtained we prefer to rely on literature to remark the unmaintained project, Khondhu et al. [KCS13] use a one-year inactivity threshold to classify dormant projects. A threshold not empirically validated, but large enough to establish the progress of the projects and arrive at a final conclusion (maintained or not).

3.3 Approach

The aim of the work is to collect data from different repositories and analyse data collected to determine labels for commits, e.g. determine the frequency of the commits made, in which time frame they are performed, in which time interval the work is mostly involved.

Previous works and tools mainly focus on establishing links from commits to issues, the number of issues or files per project, and which commits caused that bug fix. For each project, they give the possibility to browse the commit history and the issues and to inspect the links that were established [WH05a].

In such a context, we have designed and implemented an extension of such systems enabling the development of advanced data pipelines, facilitating their property and scalability, to identify and understand the level of work that the development team exhibited during the creation of a project. We relied on PyDriller, a Python framework for MSR [SAB18] capable of mining arbitrary Git repositories, to extract all core Git data, such as commits, developers, diffs, etc.

In order to compare and validate the data obtained, we needed to simultaneously analyse different repositories and to combine the information obtained. These

experiments aim to measure the ability to detect changes or similarities between the histograms of the analysed commits repository, taking as a model projects known for the use of agile software development. Similarity dictated by the evident presence of regular trends.

3.4 Design and Implementation

The tool has been developed in Python to offer multiple ways to interface with the rich Python ecosystem and with big data solutions. PyDriller framework allowed us to implement different type of python script that supports replicable and reproducible research based on software repository mining. The collection and analysis of detailed data can be challenging, especially if data shall be shared to enable replicable research and open science practices.

Given a list of git repository to be analysed, the work of the implemented tool is to collect data such as: authors, collaborators, date and time commits, commit messages and other information related to it, through the use of Git (GitHub) repository. Whenever the tool is invoked will always ensure the historical data analysis to the actual state of art of the repository that changes over the time. Once provided the list of remote repositories URL to analyse, the tool will temporarily clone all the repositories one at a time, after completing the analysis of the previous one. It navigates each commit starting from the oldest one. In this way, not only the historical evolution of the code is taken into account, but also all the background such as for e.g. documents, files and possible references, analysing the entire project.

The aim of the tool is to research and identify possible common periodic events such that there is a possible correlation between the projects analysed by navigating the Git history, or even correlation with agile development methods and process (whether they are respected or not). This allows us to identify and categorise the repositories examined and to measure the effort employed in their implementation.

The experiments ran in a cloud setting, where the tool was deployed using Docker. The implemented tool performs all the work that goes from extracting data from Git repositories to their final analysis in single-thread. A multi-thread solution can be implemented by splitting the workload to be extracted and analysed e.g. by date, week or month and process them in parallel.

3.5 Metrics Experiments

The repositories subject of my tests and analysis have been found among the trending ones in GitHub. Six metrics have been implemented. The comparison between the different metrics serves to describe the main development phases of a repository, based on the time data progression (analysing the evolution according to daily, weekly and annual steps). This section describes the proposed metrics and the experiments performed on two open source repositories, RepoDriller (Java framework) and RxJava (Java library), for computing the metrics and showing their usefulness.

The repositories were chosen in such a way that they were developed by separate teams, and have different characteristics and application domains, especially in terms of number of developers and number of classes. The aim was to cover repositories using different technologies, and explore both traditional and newly emerging projects.

3.5.1 Commits per Day of the Week

The metric *Commits per Day of the Week* (CDW) aims at evaluating the number of commits per day of the week (Mon-Sun). Therefore, it allows us to determine which are the most labour-intensive days. You might notice the workload distribution throughout the week, if there are more challenging days than others, if the histogram curve draws more slopes towards the end or the beginning of the week.

FIGURE 3.1: The number of *Commits per Day of the Week* (CDW) for RepoDriller (left) and RxJava (right).

Therefore, some characteristics can stand out and be further analysed to take countermeasures, such as e.g. distinguish the days with more or less workload, check why the distribution of commits could not be uniform, etc.

Figure 3.1 shows CDW for two repositories: RepoDriller and RxJava. Firstly, we can see that the number of commits for RxJava is much higher than for RepoDriller (15 times higher), and while the first presents a high variability of number of commits for each day, the second is more uniform for the working days of the week. Secondly, for RepoDriller the average commits per weekdays is 51.2 and for weekends is 53.5 (almost the same average); whereas for RxJava the average commits per weekdays is 1239.4 and for weekends is 510 (three times on weekdays than weekends). Therefore, for RxJava there is a greater difference than RepoDriller between the number of commits in working days and weekends. Such a result shows that for RxJava not overworking during weekends is more likely than RepoDriller, hence a better time management is performed for the former.

In general, when there are weekend commits in a number similar to working days commits, the project could be classified as a *personal* project, or *non-work-related* project, that is assumed to be exclusively in an intra-weekly period, but it's more in the private sector and then it is perhaps a side project. Still, for some exceptional weeks if the difference is not as marked it could be that the team is running late and it is catching up for a release.

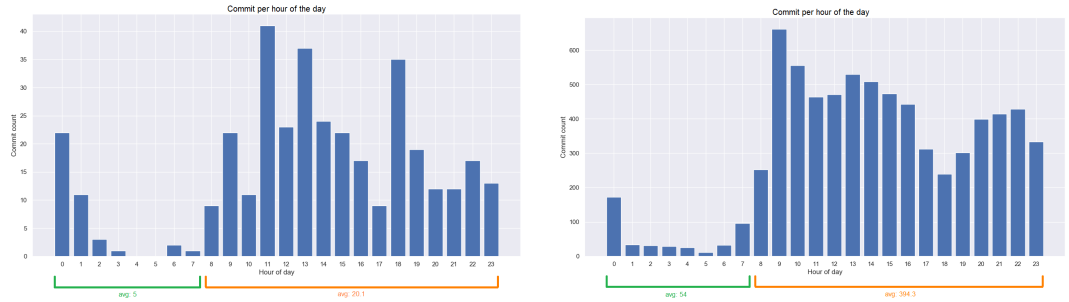

FIGURE 3.2: The number of *Commits per Hour of the Day* (CHD) for RepoDriller (left) and RxJava (right).

3.5.2 Commits per Hour of the Day

The metric *Commits per Hour of the Day* (CHD) consists in determining the commit times during the day (0-23). The possible outcomes would be to determine the usual working hours of the development team, which could be on day time if it is a *Full-Time* project, or even in the evening and at night if it is a totally *personal* project.

Figure 3.2 shows CHD for the two repositories: RepoDriller and RxJava, respectively. It can be seen that RxJava presents a higher difference between the number of commits during the day than in the night, compared to RepoDriller. For RepoDriller the average commits during the day (from 8am to 11pm) is 20.1 and during the night (from 12am to 8am) is 5; whereas for RxJava the average commits during the day is 394.3 and during the night is 54, therefore for RxJava there is a greater difference than RepoDriller between the number of commits during the day than the night. For RepoDriller, during the day there are 4 times the commits than the night. For RxJava, during the day there are 7 times the commits than the night.

Moreover, in a mainly open source project it would be possible to find commits in the early morning due to the time-zone of those who are participating in the collective project. Still, it is possible to determine the hours when there are almost no commit. In the end, the aim is to recognise whether there is a uniform effort during the day or the hours of greater tension or work.

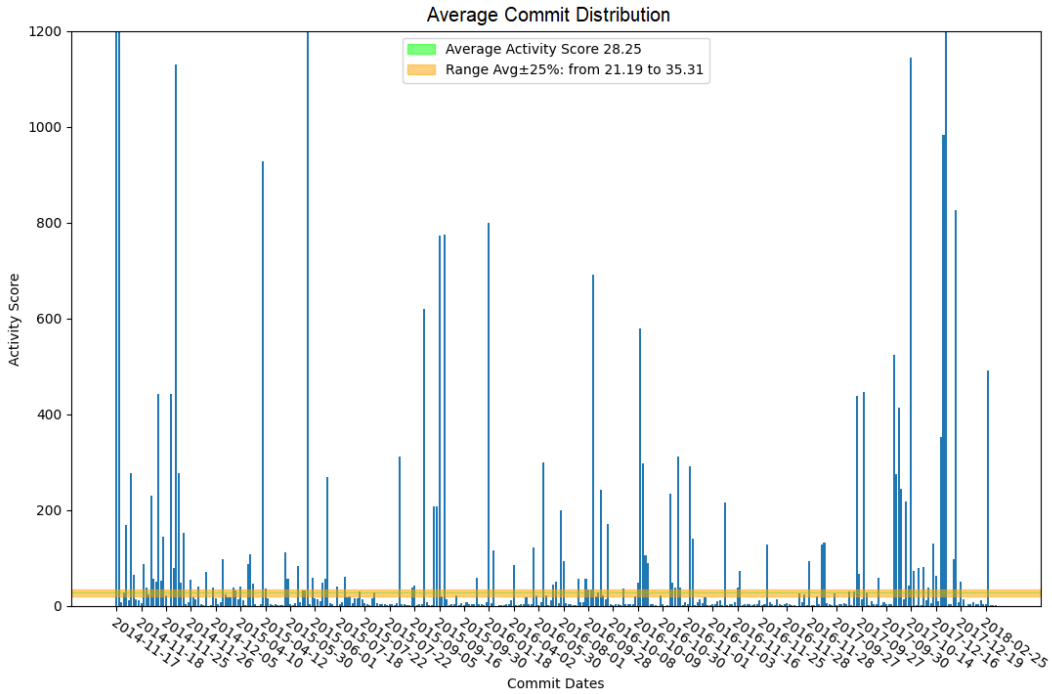

FIGURE 3.3: The use of *Average Commit Distribution* (ACD) to highlight (in yellow) the commits with an average activity score for RepoDriller (plot has been cut to an activity score of 1200 to ease data visualisation).

3.5.3 Average Commit Distribution

The metric *Average Commit Distribution* (ACD) consists in evaluating the "average" commits of the whole project. A commit is considered average if its activity score (sum of the added and eliminated lines) is between -25% and +25% of the average activity score. The average value is calculated by excluding 10% of samples from both extremities (the smallest and the highest activity score values that could negatively influence, and then invalidate, the calculation of the average), thus the mean is calculated on the 80% of samples, which is more accurate.

Figures 3.3 and 3.4 represent the activity scores of commits along the y-axis, while on the x-axis there are commits arranged by activity score (the blue histogram) respectively for RepoDriller and RxJava. Figure 3.5 highlights only the days where the average activity appears.

It is important to note that what really matter is the frequency of the similar commits spread in the project, when we find very frequent occurrences, let us

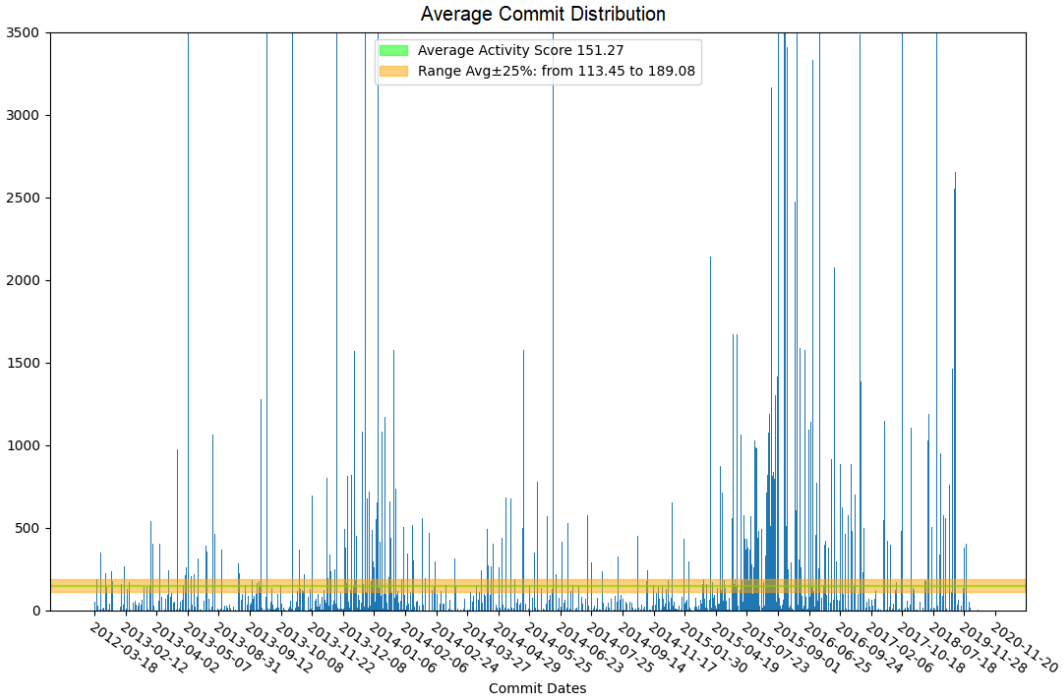

FIGURE 3.4: The use of *Average Commit Distribution* (ACD) to highlight (in yellow) the commits with an average activity score for RxJava (plot has been cut to an activity score of 3500 to ease data visualisation).

say daily basis, we can state that this project's activity is stable.

The vision of this analysis is on the whole evolution of the development project in order to verify if the study behind the repository has been constant over time and not taken up again and worked in an irregular manner.

The analysis concerns the evaluation of whether the project over time has a uniform commit distribution. This makes us understand whether developers

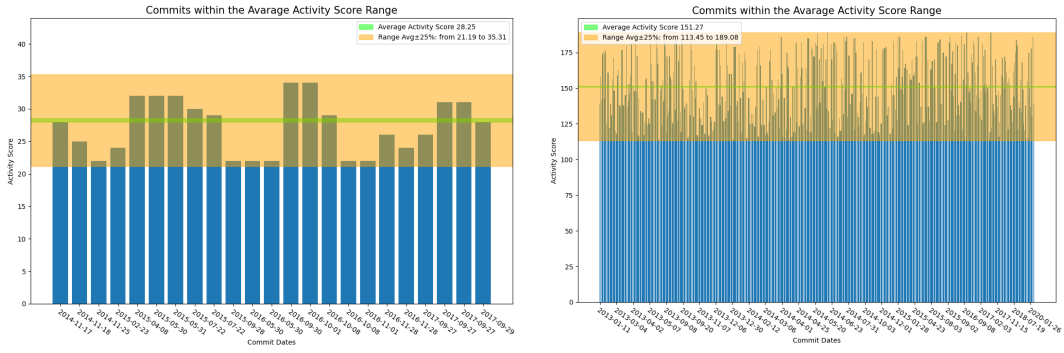

FIGURE 3.5: Distribution of commits having an average Activity Score, respectively for RepoDriller (left) and RxJava (right).

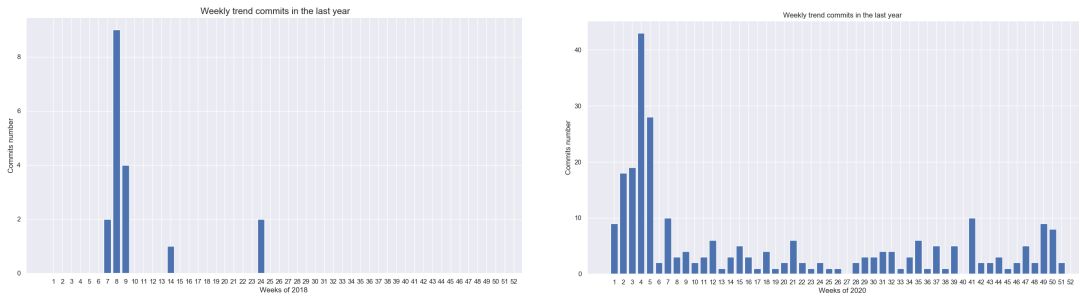

FIGURE 3.6: *Commits per Week in the last Year* (CWY) of activity: 2018 for RepoDriller (left) and 2020 for RxJava (right).

contribute in a continuous and regular way to the project, which is completely different in a project taking place only on weekends.

For instance, when preparing a newer release, intensive activities are typically performed, ensuring that the software product exhibits high quality. The files updated during development need to be verified and consolidated to ensure that changes will not negatively impact the quality of the software system [Tei17].

The aim is to understand the related information to the progress and correlation of these commits over time, to check how stable the development is: if it has a constant trend of commits or behaves like a weekend Side project.

3.5.4 Commits per Week in the last Year

The metric *Commits per Week in the last Year* (CWY) consists of a weekly vision of the progress of the last year of the project. According to the area in which the project is taking place, the last year could be an intense period for its completion. In fact, unlike a project of a very small development team, in very large and complex projects there starts to be, in this period, possibly a whole series of tests and verification of its correctness, safety and robustness, even though there could be possible unexpected changes last minute due to the outcome of the tests or due to company and customer's end decisions.

The analysis is performed from January 1st of last year when the last commit was made, from that day the progress of the commits performed is evaluated in the following 52 weeks. Observing the number of commits in the last year,

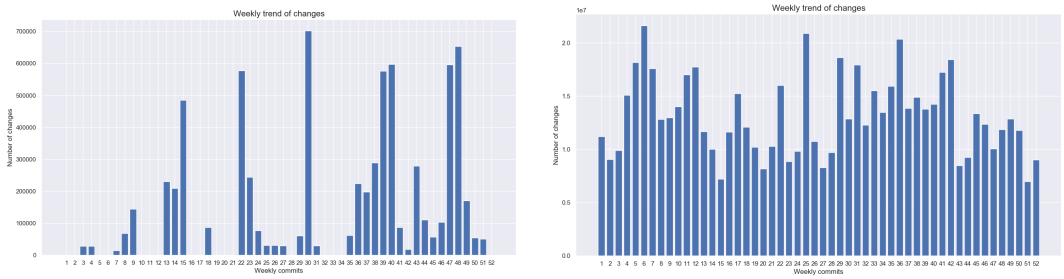

FIGURE 3.7: *Changes per Week Trend* (CWT) for RepoDriller (left) and RxJava (right).

it is possible to gain value for the metric both when the project is in progress or when it has ended.

Figure 3.6 shows CWY for RepoDriller and RxJava. As we can see, RepoDriller has less commits and less uniform effort during development time. RxJava presents a more sustained commitment in the initial five weeks and then an pretty uniform effort for all the other weeks of the year.

3.5.5 Changes per Week Trend

The metric *Changes per Week Trend* (CWT) determines the trend of the overall source lines of code from the beginning of the project, it is a weekly analysis of the entire developing of the project but starting from the first commit, in order to always evaluate the effort trend throughout the program creation phase.

This analysis provides the possibility to monitor the progress of the project by filtering the repository on specific file extensions, these files will be opened and the number of rows added or removed counted from time to time in each commit.

The aim of this metric is to show how development work has been distributed over the year. It would be possible to recognise the weeks exhibiting intense work, due to deadlines (e.g. close to release dates) or commits on holiday weeks. A Full Project would likely not be under active development during holidays, whereas on the contrary, holiday time could be dedicated to the development of a Side Project.

Figure 3.7 shows CWT for RepoDriller and RxJava, for each of the weeks in a year. For RepoDriller there are bursts of changes in a few weeks, than very little or no commits in other weeks, hence the workload is not fairly distributed over time. RxJava presents a much more sustained development (much higher number of commits) and an almost uniform distribution of workload over time.

3.5.6 Lines Of Code in Time

The metric *Lines Of Code in Time* (LOCT) determines the trend of the overall source lines of code from the beginning of the project; it analyses the progression course of LOC in the repository in order to evaluate the important phases in the development of a project.

This size metric gives the possibility to monitor the program length from time to time in each commit. The number of lines in the commit files reaches very high values, especially in reference to a corporate project that had code refactoring in the long run (a technique for modifying the internal structure of portions of code without changing its external behaviour), applied to improve some non-functional features of the software.

The aim of this metric is to show the rate of change of the project size. It can reveal if this has been done regularly and in conjunction with what particular events: code refactoring, integration and adoption of new packages and library, new development tool that has radically changed the initial project structure, or the continuous integration of new feature requests increasing work and effort to be devoted to project development. Identify the periodic trend of these factors best characterises the historic development of the project as the commitment of the team that worked on it.

Figures 3.8 and 3.9 represent the lines of code trend in each commit of the project along the y-axis, while on the x-axis there are the commit date made respectively for RepoDriller and RxJava. On RepoDriller, projection lines were drawn to show periods of more intense work activity. Both projects, show an increase of lines of code, with some particularities that mark each of them.

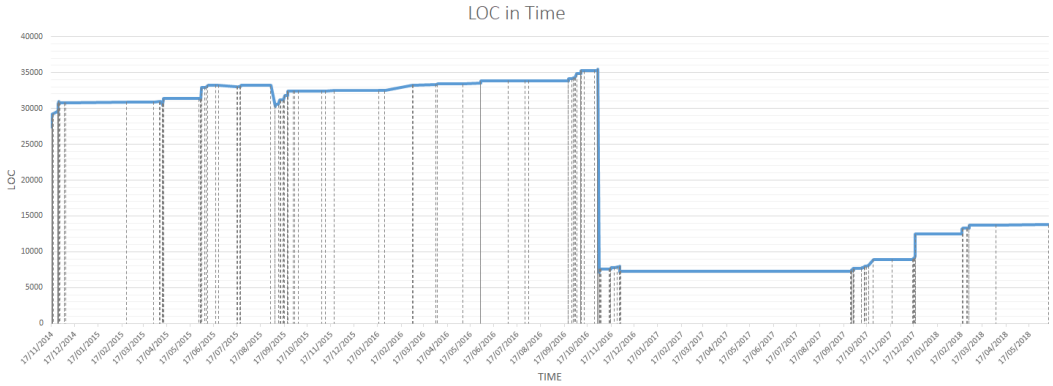

FIGURE 3.8: The use of *Lines Of Code in Time* (LOCT) to highlight the line of code trend for RepoDriller.

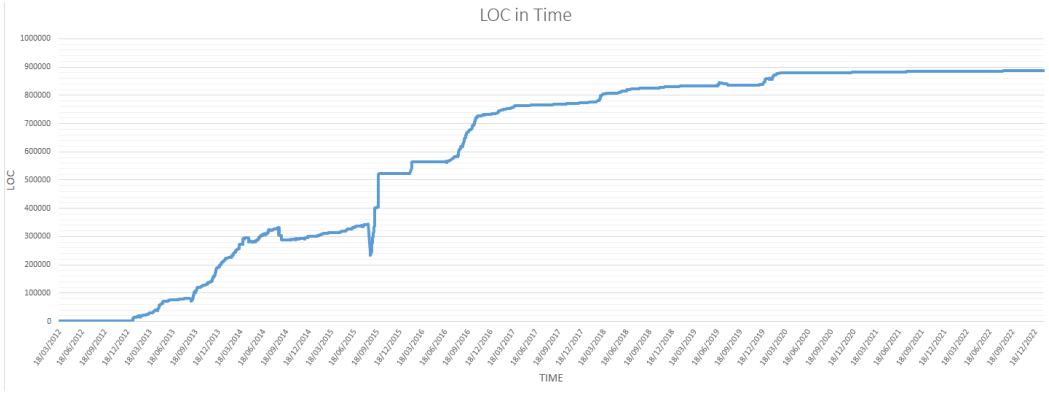

FIGURE 3.9: The use of *Lines Of Code in Time* (LOCT) to highlight the line of code trend for RxJava.

RepoDriller has a much shorter development time than RxJava. It shows slow steady growth in the first two years of development until the end of 2016 with an evident drop of 80% of LOC due to an evident code refactoring. After this specific period, LOC increases week by week in a regular way, but less rapidly than RxJava.

In RxJava, after a quite constant growth, we can see in the second half of 2014 a reduction of about 44000 LOC such as a much more drastic reduction of about 105000 LOC in the same period one year later (2015) with a subsequent very fast LOC increase in the following days until it stabilises completely nowadays. As reported by RxJava documentation the reason for this change is due to a change of version of the tool (2.x). The purpose for 2.x was: leverage Java 8+ features, Reactive Streams compatibility and gain performance through design changes.

Excluding some particular events, in the projects we examined there is an expected growth, in terms of lines of code.

3.6 Conclusions

We have proposed to analyse software repositories by means of metrics that capture the effort of developers and their practices. Such elaboration has allowed us to highlight and to distinguish times of intense and less intense job, to determine the level of work constancy of developers and finally to characterise if the project is still in development phase (maintained) or not.

Six metrics, i.e. Commits per Day of the Week (CDW), Commits per Hour of the Day (CHD), Average Commit Distribution (ACD), Commits per Week in the last Year (CWY), Changes per Week Trend (CWT), Lines Of Code in Time (LOCT), have been used to analyse two repositories and have shown that the practices adopted are different.

For the first, RepoDriller, a small project, there is no significant difference between the workload during weekdays and weekends. Moreover, the workload is performed in bursts rather than continuously during the year. The other

project analysed, RxJava, is a much bigger project, and the work is mainly performed during weekdays and continuously during the year. The proposed metrics swiftly capture some of the development practices under investigation.

By using the above process metrics, projects could be grouped in three categories: Open Source, Side Project and Full Time Project. We evaluate and find the differences that characterise them, such as: how many people are working on it, how long the project is kept going, whether the project is large or not based on the number of releases, etc.

It has been shown how with PyDriller it is possible to extract information from any Git repository such as commits, development teams, modifications, diffs and source code, allowing to help researchers, scholars and professionals who perform MRS.

The tool that was created with PyDriller can, given the definition of a theoretical metric deduced by the developers of mining software repository, validate it with appropriate tests. Each of these metrics has its own meaning and precise course. PyDriller allows you to validate this type of analysis and to draw the appropriate considerations on various projects, including large-scale ones. Each metric used on this tool determines results that can be used to provide evidence for relevant conclusions.

With these analyses, researchers can empirically investigate, understand and discover useful information for effort on software engineering. With these pieces of information, the development teams will be able to take actions and decisions to improve their code and development process.

Chapter 4

Discover Scrum Model for Agile Methodology

The coordination and teamwork are fundamental aspects in a software development team. The collection of principles and good practices has made Agile Software Development Methodologies increasingly popular among software companies. With its adaptive nature and high flexibility, Scrum is an agile framework able to manage and control the software and product development process. Achieving an objective performance measurement in ASD addresses certain challenges due to different team performances and project complexity. It becomes important to know how to properly integrate effort estimation with good agile development practices. In this chapter, novel metrics will be proposed to reveal the adopted development practices by analysing data extracted from repositories. The utility of the proposed metrics is to reveal the categories of activities performed during the development of the project, the effort trend of developers, to detect Sprint week and possibly suggest improvements in the adopted Scrum practices.

4.1 Improving Effort estimation on ASD

Effort estimation is one of the critical and open challenges software engineering activities performed during the development process. It is defined as the method by which effort is evaluated. The estimation is known as the amount of

resources required to complete a project activity in order to deliver a product or service that meets the given functional and non-functional requirements to a customer [Fer+20b].

Accurate effort estimation is an essential factor for planning software projects and with different types of development processes, there are different perspectives on planning and estimation. Various software metrics have been proposed in previous study, which focuses on measuring the development performance with some of the common techniques for deriving an estimation in ASD: expert opinion (estimate provided by expert intuition), analogy (task size comparison), disaggregation (splitting task into smaller), planning poker (consensus-based estimation technique that's the combination of previous approach) [Coh05].

For more than 20 years since its official publication, due to its peculiar properties and consolidation in IT companies, the Agile Software Development Manifesto inspired the introduction of new methodologies in software development field. A large number of different approaches to software development have been submitted, of which only a few have survived to be used today [Abr+17].

Each method has its own principles, life cycle, roles, advantages and disadvantages. All of these agile software development methods build the software in iterations and incremental processes. The estimation needs to be done progressively especially because most of Agile methods involve short-term planning for every little step of changes made, just like release planning, iteration planning and the current day planning [AGH17; MKK09].

Effort estimation is a complex operation. Despite the vast number of approaches, the accuracy of software effort estimation models for agile development still remains inconsistent [PMR17; Fer+20b]. R. Popli et al. identify which issues affect the effort in an agile system [PC14].

- Effort estimation. The estimations are done incorrectly in units of time regardless of the breaks that employees take in those time units (meetings, lunch breaks, checking email, phone calls etc). It is more important to estimate how much each member can spend for Sprint related work.

- Release Date estimation. The release date of the final product is often set without considering various factors such as velocity and cost benefit.
- Release Risk tracking and estimation. Risk estimation is not done in Agile Estimation. It is necessary to track total risk values of a project, especially when an estimate risk of deviation is high.

Researchers have the burden to gather, investigate, validate evidence from existing studies and discover useful information by analysing various undertaken development process to build a corpus of knowledge and improving the quality and productivity of the team. Mainly because if the effort estimations are accurate, they can contribute to the success of software development projects, while incorrect estimations can negatively affect companies' marketing and sales [AG18].

That is the reason why development effort estimation is central to cost estimation in software development and the most difficult parameter to estimate [ARG06]. It is becoming increasingly important to integrate effort estimation with good agile development practices. Research in this area has found wide application and helps companies teams and organisations to evolve and perform more efficiently. Due to the invisibility of software development activities it is not easy to check if members of a development team are actually performing the efficiency of agile approaches [CA16]. A possible solution can unlock potential to make better company choice and invest time and money appropriately.

To facilitate project planning and for the eventual successful implementation of the project, software developers require effective effort estimation models [Unt+11]. Software metrics play an important role to measure aspects, features and differences that characterise the projects. The status and productivity of the team, how large is the project based on the number of releases and Sprint, how many people per team are needed to do the tasks, are just some of the aspects that allow companies to understand and change the behavior of their team and consequently define new rules [KFW18]. Project Manager and the development team can use the proposed metrics to monitor the progress and improve the accuracy of programmer effort estimation during the project lifecycle. Such

metrics allow us to identify characteristics and the effort in software projects and consequently the developers behind them.

The rest of the chapter is organised as follows. Section 4.2 presents some principles and structures of Scrum framework. Section 4.3 describes the related work. Section 4.4 discuss the importance of estimation process on Sprint. Section 4.5 introduces our approach and implementation. Section 4.6 presents the proposed metrics. Finally, Section 4.7 contains the conclusion.

4.2 Scrum framework structure

Scrum is one of the most commonly used agile methods that provides steps to manage and control the software and product development process. It is an empirical approach that applies the ideas of industrial process control theory to systems development [SB02]. Scrum concentrates on how the team members should function in order to produce the system flexibly in a constantly changing environment.

The Scrum main idea is that systems development involves several environmental and technical variables (e.g. requirements, time frame, resources, and technology) that are likely to change during the process. This makes the development process unpredictable and complex, requiring flexibility of the systems development process and to be able to respond to the changes [Abr+17].

First described by Ken Schwaber in 1996, Scrum was designed: to better split large task items into manageable smaller works, to boost development speed, to coordinate and improve individual improvement, to increase performance and work rhythms, to provide continuous support to stakeholders and to have good communication of performance at all levels. It promotes adaptive planning, progressive development and delivery. Scrum defines an iterative approach and encourages very rapid response to change [SBS17]. Aspects and properties that helped to make Scrum a framework that achieves greater flexibility, higher-quality products and customer satisfaction.

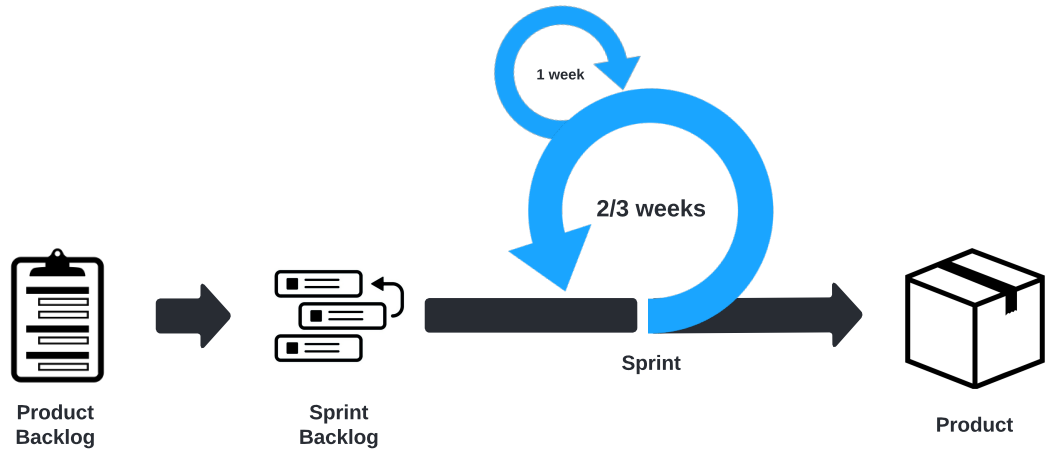

FIGURE 4.1: Single Sprint process: from the selection of tasks to be implemented (Backlog) to a potential product delivery (Product)

The main core of Scrum is Sprint. Sprint is a regular iterative time period (cycle/iteration), usually lasting two to four weeks, in which a small team works on assigned and unchangeable tasks to create a potentially releasable product. Figure 4.1 depicts the sequential process of a single Sprint. A new Sprint only begins once a previous Sprint has been concluded. Team members choose the tasks they want to work on and begin development. Each Sprint includes the traditional phases of software development: requirements, analysis, design, evolution and delivery phases. The architecture and the design of the system evolve during the Sprint development.

Sprints are conducted by a cross functional and self-managing Scrum Team. Three different roles make up the team: the Product Owner, the Scrum Master and the Developers. Each member of the Scrum team is an expert in various fields and has different responsibilities to achieve the final Product Goal. The Scrum Team, consisting of developers, quality assurance engineer and a documenter, is responsible for all product-related activities from stakeholder collaboration, verification, maintenance, experimentation, research and development.

Team members consist of developers and testers who create and adapt plan for the current Sprint Goal (reminding developers why the tasks are being performed and at which level of detail to implement them). The Product Owner

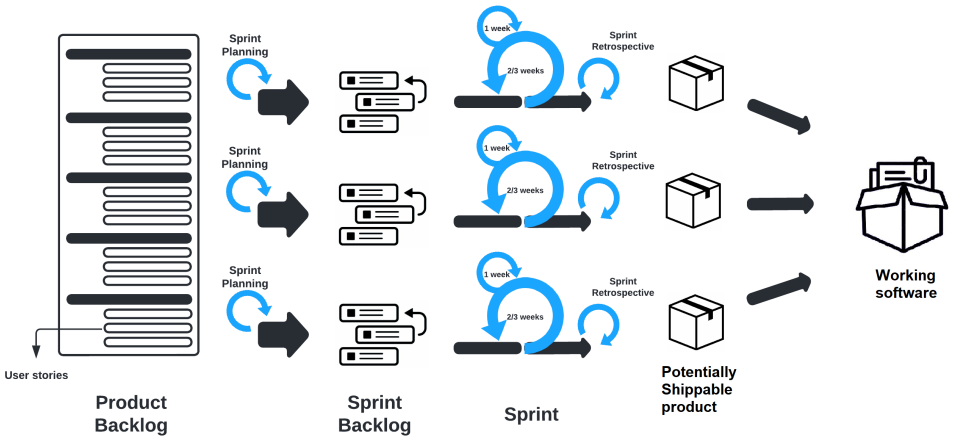

FIGURE 4.2: Scrum framework process overview

is the owner of the project contract, he also manages the software specification and the Product Backlog. In order to achieve set goals, the Product Owner ensures that the Product Backlog is transparent, visible and understood by the Team. The Scrum Master is a team leader. He is an essential link between Product Owner and project Team for the management of the Product Backlog and definition of Product Goal. The Scrum Master aim is to remove any impediments on the Scrum Team's progress (create an appropriate and inspiring work environment for the team and provide team members with the information they need to perform their tasks), keep the team working as productively as possible to respect the prefixed time schedule and verify the implementation of principles and rules of the framework, intervening if they are not respected.

The task for a Sprint is decided by the Sprint Backlog. Sprint Backlog is a set of requests, originating from the customer and understandable to Developers, usually called User Stories, to be achieved in a single Sprint. The requirements are prioritised and the effort needed for their implementation is estimated. Backlog items can include, for example, features, functions, bug fixes, defects, requested enhancements and technology upgrades.

During the pre-sprint planning, as shown in figure 4.2, features and functionality are selected from the release Backlog and placed into the Sprint Backlog. The Sprint Backlog is nothing more than the further breakdown of the elements of

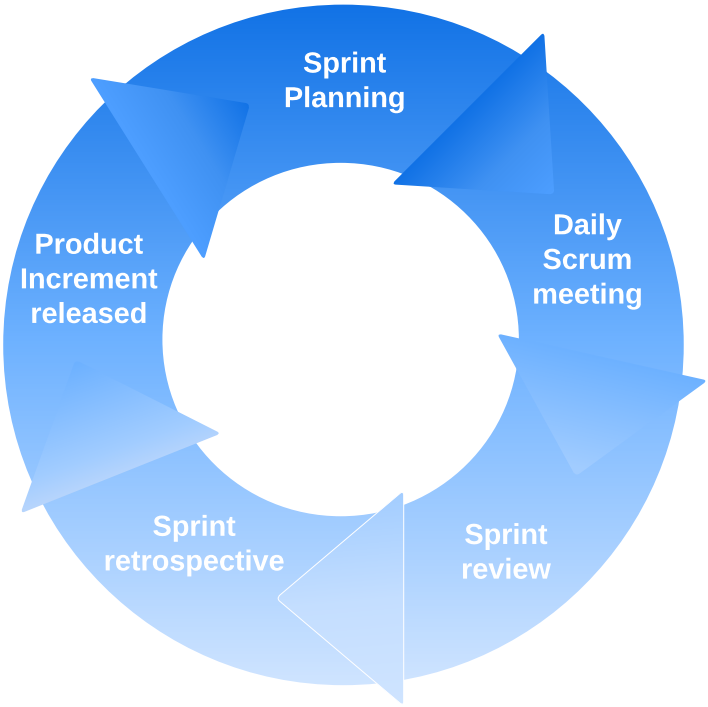

FIGURE 4.3: Figure depicts the Scrum lifecycle

the Product Backlog. It is a list of all requirements, determined by the product owner, that the end product must meet. All items on the Product Backlog are detailed by a description, order, and size. It is possible to keep track of how much has been completed in order to indicate progression and monitor the total work remaining. The Product Backlog list is constantly updated with new and more detailed items, as well as with more accurate estimations and new priority orders. At every Sprint iteration, the updated Backlog is reviewed by the Scrum Team to gain their commitment for the next iteration.

Once the requirements are completed, no more items and issues can be added nor can any new ones be created. The system is ready for the release including the tasks such as the integration, system testing and documentation.

Following "The Scrum Guide" by Sutherland and Schwaber, there are four events in Sprint which constitute the Scrum framework (figure 4.3). Sprint meeting organised by the Scrum Master to create a cyclic succession on the software development process.

- *Sprint Planning* Starting point of a Sprint. It's a team meeting discussion

where an effort estimation poll establishes the product backlog items to be undertaken into the Sprint (Sprint Backlog). The poll shall be carried out taking into account the priority and the content of tasks to perform.

- *Daily Scrum Meeting* In this brief meeting, the team member inspect the progression work applying changes to the plan if necessary and also serve as planning meetings: what has been done since the last meeting and what is to be done before the next one. Daily Scrums improve communications, identify impediments, managers on the status of the project, promote quick decision-making, keep the entire team focused on a common goal and consequently eliminate the need for further meetings.

- *Sprint Review* On the last day of the Sprint, the Scrum Team demonstrates the result obtained in the Sprint in presence of all stakeholders. The customer can provide feedback on the progress of the project and make decisions about the following activities.

- *Sprint Retrospective* The Sprint Retrospective concludes the Sprint. This meeting aims to introspect the ongoing Sprint. The purpose is to identify and understand the source of the problems encountered with the aim of increase quality and effectiveness.

- To deliver a potentially releasable product, at the end of each Sprint the Team demonstrates the progressive feature increments to the product owner. A meeting is held to analyse project progress and demonstrate the current system.

Each Scrum step guarantees certain properties that are essential to achieve results[1].

A low level of Transparency can lead to decisions that reduce value and increase risk. Instead in Scrum a transparent working process facilitates the inspection of every step.

Frequent and diligent Inspection of the agreed targets' progress allows for the detection of undesirable deviations or problems.

[1]Sutherland and Schwaber https://scrumguides.org/scrum-guide.html

Inspection enables Adaptation. If the result product is unacceptable it shall be adjusted in accordance with the requisite requirements.

Making the whole development team aware of the problems and changes made significantly lowers the management costs. Constant communication with stakeholders increases the quality of the final product. Small working teams maximise communication and minimise overhead. Constant testing and documentation of a product as it is built.

The attention and rigor of the aforementioned Scrum steps try to avoid wrongs dynamics or results such as: code quality issues due to time limitations, missing integration testing not performed properly, no bug free code under working pressure, input of customer requests in an already begun Sprint, not respect Sprint timing, disparity in the development team, absence of a scrum training and more [RM12].

4.3 Related Work

There are many studies about ASD estimation models in literature. Some of the well-known studies in literature are briefly summarised below.

Alostad J. M. et al., build a fuzzy based model that can improve the effort estimation in Scrum framework. The model simulates the role of Scrum Master and development team in effort estimation during the sprint planning phase. For each task, the researcher uses Scrum Story Points as a measure of effort estimation in Scrum projects. A final Estimation Accuracy (Over Estimate, Well Estimate, Under Estimate) determines when it has deviated from the initial assessment [AAA17].

A prediction model that uses developers' features to estimate story points to issue reports in open source projects was build by Scott et al [DLR10]. A supervised approach has been applied with a publicly available dataset used in several studies, it consists of issue reports of eight open source projects. For a final validation, they compare the results of several models that are trained with a different set of features.

Fernández-Diego M. et al., present a systematic literature review to characterise estimation activities in agile projects. The paper collected and review the effort measure, estimates and predictions of 73 papers from 2014 to 2020. These studies have been analysed with the purpose of comparing the models used in terms of accuracy. The effort estimation methods have been used in six different agile methods (Scrum included). What is clear from the studies is that the most used estimation technique is Planning Poker with a frequency of 25% followed by estimation methods that rely on Expert Judgement at 11% and Wideband Delphi at 5%. The most applied models in the data analysis are Machine Learning at 20%, Neural Network at 17%, Functional Size Measurement at 16% and Regression model at 8%. The contribution of this paper is to report the state of the art in the field of effort estimation in ASD by means of a systematic literature review [Fer+20b].

Built on existing work, Alhazmi A. et al., implemented a project decision support and planning system in Scrum methodology called Sprint Planning dEcision Support System (SPESS) tool [AH18]. SPESS takes into consideration the adoption of consolidated planning poker with the integration of more features: developer competency (knowledge and skills that the developers have to work on a specific task), developer seniority (different skill level implies a different timing in the problem solving on the same task) and task dependency (relationship between two tasks, where one task cannot start implementation until the other task is completed). The main task of SPESS is to guarantee that each team members contribute to the fullest of their potentials assigning the tasks of each Sprint to developers adequately and to optimise project planning for the shortest possible time.

As shown in the aforementioned literature review, most of the existing works in the domain of effort estimation are based on planning models. However, at least for open software projects, these models can be improved with the use of several additional data sources. Mining Software Repositories has the ability to use process mining techniques to extract knowledge from a variety of information sources. A promising line of research is the use of this field to

analyse the rich data available in software development repositories in order to search and define metrics that capture Sprint and developer activity, which can in turn be used to better estimate effort in a project.

4.4 Relate Sprint trend in Scrum models

Over the years, Scrum has gained the reputation of being the approach that accelerates software development and better addresses customer feedback through continuous iterations. The method is characterised by a short iterative development life cycle, testing, frequent deliveries and meetings that provides steps to manage and control the software and product development process. Not for nothing Sprint is the core of Scrum process.

Sprint lasts from 1 to 3 weeks or more. While Schwaber originally suggested sprint lengths from 1 to 6 weeks (Schwaber, 2002), duration is commonly held at 4 weeks. In this time frame, a small team works on assigned task through a full software development cycle including planning, requirements analysis, design, coding, and unit testing. The last Sprint is dedicated to testing, on which any reporting bugs must be promptly fixed by the end of the week. This periodicity of events lets the project adapt to possible changes quickly. Because the aim of each Sprint is to deliver a potentially shippable product [SBS17].

In the Scrum process, there are several events performed on each Sprint that have a significant influence on the effort estimation in order to monitor team performance. As mentioned before for most agile models, these factors are: Development Team Experience, Task Complexity, Task Size and Estimation Accuracy [AAA17; Usm+14]. Having an inefficient development team who doesn't have the required experience for some tasks would reduce the developers' competences to manage deadlines and lower the quality of the final product [AH18]. This would slow down the work schedule to be completed.

This is a good starting point to study the behavior of the teams involved. Aspect that has more and more impact when it addresses to open source developers. Company records and logs typify proprietary software from which research can

be established and made. Instead, the effort can not be easily tracked on open source repository for the reason that such records do not usually exist [ARG06].

After all, the estimation of well consolidated agile software methods depends on an expert opinion and the presence of project's historical data. In absence of data to be analysed and expert judgement, the previous method like analogy and planning poker are not useful [PC14]. A perfect example are the open source repositories. During the latest years, free and open source software has gained a lot of attention from the industry. Following this interest, the research community is also studying it. This chapter intent is to investigate this case study: to detect the presence of Scrum framework use on any type of repository (open source included).

A few metrics have been implemented to identify a possible thresholds to discover Scrum presence, which varies from one project to another. A metric tries to better identify the time windows from which it can be inferred and better characterise the performance of each Sprint. A BoW metric has been implemented to better categorise the type of Sprint analysed. At the same time, it is also possible to highlight new aspects and considerations: to determine which branches that compose the repository have most characterised the Scrum and why, the number of actual developers who contributed to the creation of Sprints, an automatic graphical detection and distinction of Sprint development weeks from those of testing/debugging, what are the most labor-intensive Sprints or determine whether the deadlines have been unable to comply.

This estimation process is worth to be studied. Therefore, our automatic approaches, that aim to support developers in the estimation process, are implemented to support these analysis.

4.5 Approach and Implementation

The aim of the work is to provide companies in the sector, and not only, a tool to validate the application and use of the well established agile approach Scrum. We offer the possibility to analyse local and remote repositories by

collecting data to categorise relevant git commits, e.g. determining the frequency effort of commits weeks, if the commits' timeframes are constant over time (without excluding weeks), or in which time interval a different gap of commits is highlighted, to discover possible regular trend of Sprint in the project history.

A literature review by Kurnia et al., grouped and discussed 34 Scrum software metrics. From that classification, previous work focuses mainly on the study of metrics to be applied during the meeting phase: sprint planning, sprint review and sprint retrospective [KFW18]. As well as, the most popular native effort estimation method in Scrum, Story Points comparison [AAA17; DLR10]. Story Points represent an estimate of the overall effort required to fully implement a task. It's a combination of time needed for each task to be analysed, developed, tested and implemented in the final product.

With this in mind, to diversify the field of research, we have elaborated and implemented a couple of metrics to expand tools and knowledge in this area. The benefit consists of integrating these techniques with the already consolidated tools for detecting Scrum Sprints without incompatibility issues and to understand if the development team follows the dictates of a Scrum framework. We relied on PyDriller, a Python framework for MSR [SAB18], capable of mining arbitrary Git repositories, to extract all core Git data, such as commits, developers, timestamp, etc.

In order to verify if development teams of open source projects typically use a Scrum Agile software development process, for our experiment we rely on open source projects with at least 10-16 developers per team. The number of developers associated with a project can increase dramatically if it is maintained over time. This is due to a possible generational change, new collaborators (especially in open projects), or the integration of new members into the team. The repository will keep all the history of changes made and therefore all the developers who participated in it.

To find out which open source project adopt Scrum methodology approaches, particular attention was given to the repositories of research institutions and open organisations. Among these projects, we analysed 113 repositories made

TABLE 4.1: Projects descriptions

Name/System	Start-End Dates	#Devs.	#Commits
foundation	17 Aug 2016-15 Mar 2022	130	429
landscape	4 Nov 2016-18 Mar 2022	435	3,801
tag-security	13 Mar 2018-17 Mar 2022	151	1,398
devstats	30 Jul 2017-18 Mar 2022	21	6,888
gitdm	19 Apr 2017-18 Mar 2022	692	5,245
tag-contributor-strategy	28 Feb 2020-15 Mar 2022	18	221

available by Cloud Native Computing Foundation (CNCF). CNCF is an open source software foundation that promotes the adoption of cloud-native computing. Many of the technologies that the CNCF researches and creates are hosted on GitHub[2], a restricted list of projects and their characteristics are shown in Table 4.1.

To meet the demands to understand if the same development team is performing well over time, we offer companies the opportunity to analyse the development progress undertaken by the team. We give the possibility to examine several repositories simultaneously and to combine the detected changes or similarities between the histograms of the analysed commit repositories. Positive results dictated by the evident presence of regular Sprint trends. Scrum Master and the development team can use the proposed metrics to monitor the improvement in effort estimation accuracy during the project life.

4.6 Metrics Experiments

In order to report the computed results and their degree of usefulness, this section describes the proposed metrics and the experiments performed on CNCF repositories. The metrics are aimed to identify whether the software development team has adopted a Scrum Sprint methodology. A data plot visualisation was made for each data analysis workflow. The charts show how much effort was applied for each distinct type of Sprint.

[2]GitHub Cloud Native Computing Foundation https://github.com/cncf

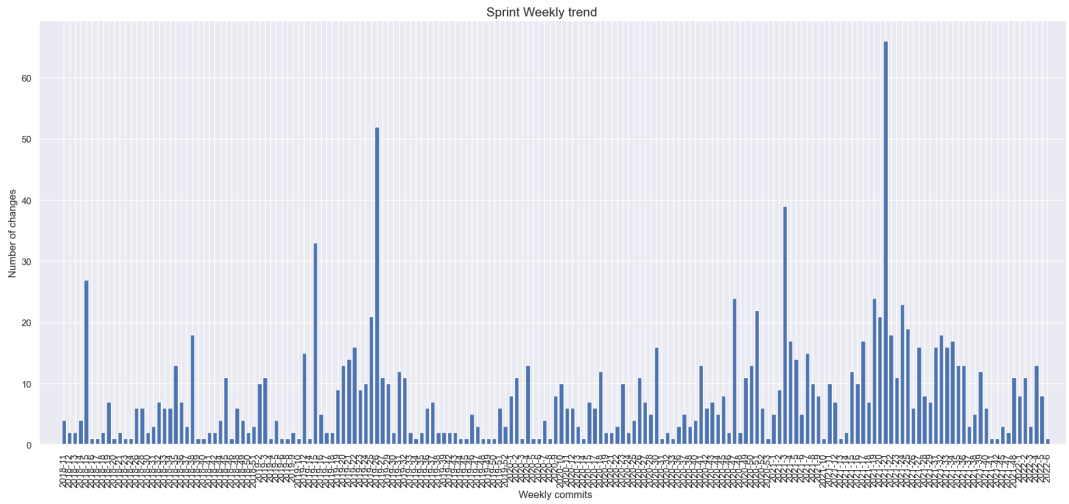

FIGURE 4.4: *Sprint week commit trend* (SWCT) on 'tag-security' (CNCF)

4.6.1 Sprint week commit trend

The metric *Sprint week commit trend* (SWCT) aims at evaluating the effort of developers. The method counts the commits per week and reports the results over time. The development of a project is divided into weeks of project progression and weeks of testing, experiments and verification. The possible metric outcomes want to identify the periods of increased traffic effort against the periods of lower. The core of the metric is to establish the different latency, and a possible effort gap during development time, that demonstrates the use of Scrum Sprint Agile software development processes. The analysis can be performed both on the total repository history or on the individual remote branches that compose the project.

Figure 4.4 shows SWCT applied to a CNCF project repository: 'tag-security'. On the x-axis is represented the Sprint week time of the project and on y-axis the number of changes the project committed per week. What can be noted is that next to the development Sprint period the activities decrease drastically or even stop as the testing activity starts. During the testing week there are numerically few commits compared to the development phase. After this, the cycle of Sprint starts again. As we expect, the outcome reports continuous periods of time where the effort of the developers is high average spaced out by single weeks of low effort. It is possible to delineate a possible cycle window

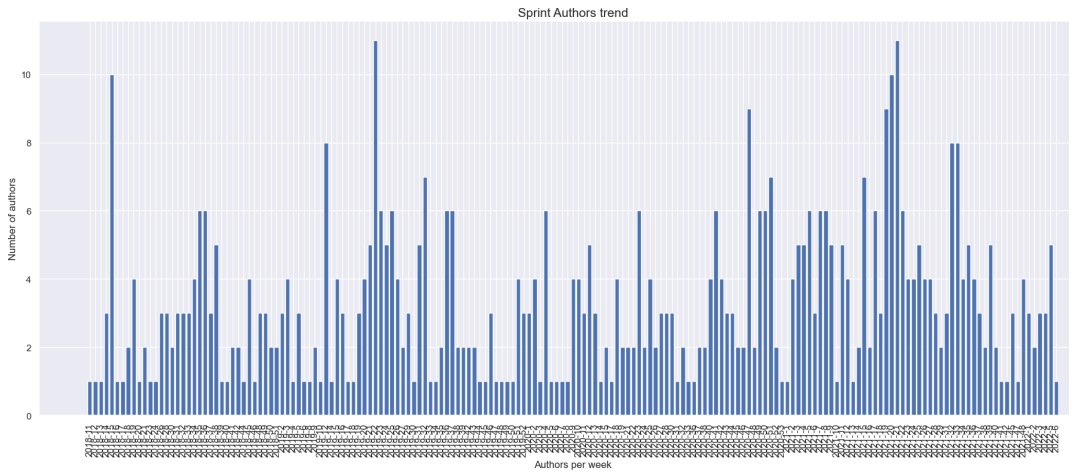

FIGURE 4.5: Number of committer in each Sprint week on 'tag-security' (CNCF)

that characterises the Scrum sprints.

To meet the requirements of Scrum Master (as shown in figure 4.5), there is the opportunity to view the actual number of authors who participated to the development of each weekly Sprint. Scrum Master can monitor the actual participation of the team and then divide the effort present in the single Sprint week, as well as he could pay attention to borderline cases e.g. an alarming fact that the management of entire Sprint is entrusted to a single developer. Figure 4.6 represents the percentage distribution of commits per author. The metric gives an extra vision to guarantee that each team member contributes to their full potential.

To better organise the project structure, developers can use the branches of the repositories to divide and schematize their work. Due to different fields and approaches, it is not uncommon to discern branches dedicated to the testing phase and branches dedicated to the implementation of features. As shown in figure 4.7, a combined analysis of all branches that characterise the repository gives a better and global view on how the project is developing. Each branch is plotted with different colour and size (main branch in blue), based on how much effort was put into it weekly. From the result obtained it is possible to detect: which branches are most present and better distinguish the repository history, if and which branches are recurrent over time and make a clear distinction of

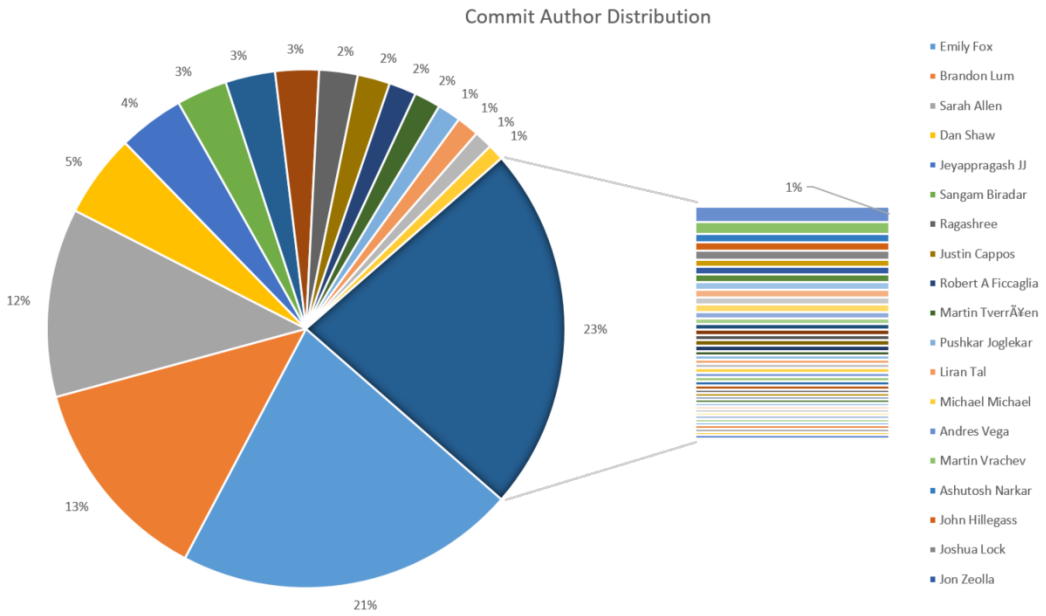

FIGURE 4.6: Percentage distribution of commits per author on 'tag-security' (CNCF)

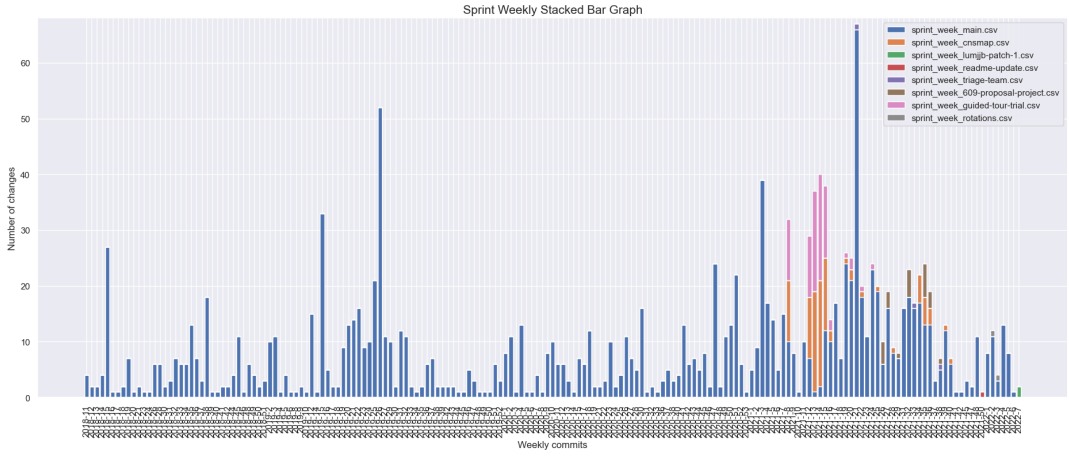

FIGURE 4.7: Sprint week commit branches trend on 'tag-security' (CNCF)

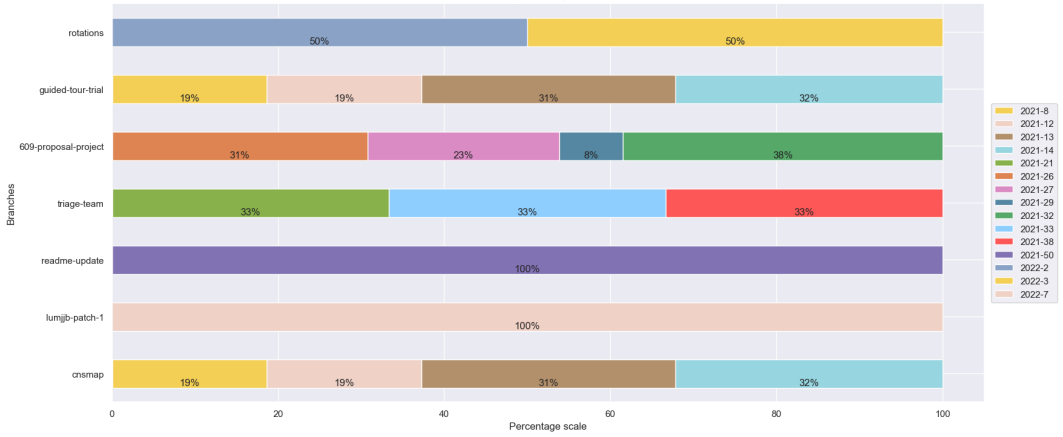

FIGURE 4.8: Percentage of effort employed in the first development weeks in all branches of the 'tag-security' project. Note cases of work done in the same period but in different branches

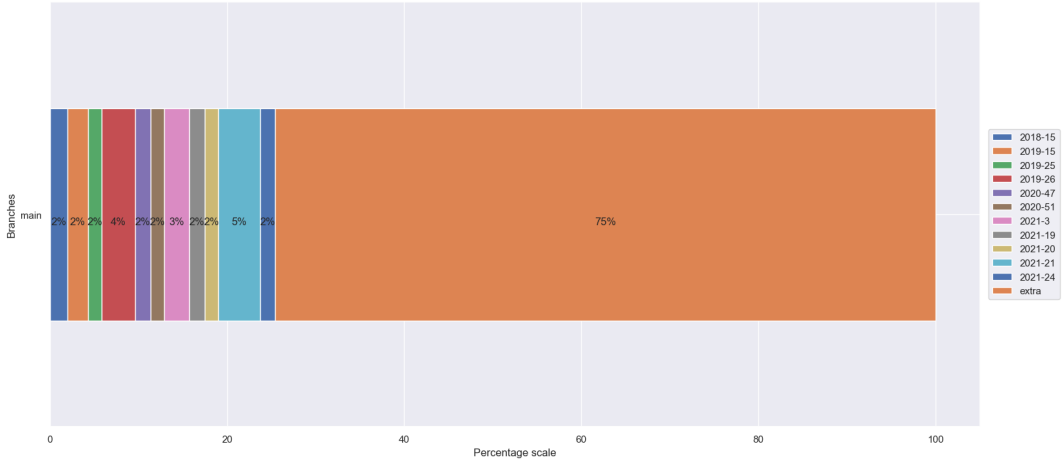

FIGURE 4.9: Percentage of effort employed in all weeks of main branch of 'tag-security' project. Extra: 75% is the sum of commits percentage with less than 10 commits per week

what category of work these branches belong to.

To have a clear vision of the contribution and effort employed in the individual branches in a weekly Sprint cycle on 'tag-security' repository, the figure 4.8 depicts the percentage of commits present in individual branches in the first consecutive weeks of sprint in comparison with sprint commits percentage in the main branch in figure 4.8. In the main branch Sprint report, it is possible to prominence note the last Sprint tagged extra in dark orange, it encloses the set of Sprints with an effort of commit not enough significant and high values.

It is possible to set the weeks (3+1) to display in all branches.

From this point of view, it is possible to understand which branches have mostly characterised the project, how much is the effort difference between the development branches compared to possible testing and debugging branches, having a visual demonstration of what kind of branches have been more laborious than the others.

4.6.2 Average Sprint Window

The metric *Average Sprint Window* (ASW) consists in evaluating any possible Sprint Weeks in the project history within a consecutive time range defined by a sliding window. An automated process has been established in order to ensure and achieve the success of the results obtained from data mining. Once the size of the "time window" is set, the data undergoes a first filtering process to satisfy the actual weekly continuity in the time frame (the search is carried out only for consecutive weeks).

A succession of Sprints is considered valid if the average effort performed by the developers (number of commits made during this period) is greater than the effort of the next week, which is assumed to be a week of testing. The window is interpreted as follows: the first n-1 weeks as development Sprint, the last Sprint (week n) as a testing and verification week. The complete method for calculating the Average Sprint Window metric is shown in Appendix A.1.

Once the window is defined, auto-filters are applied to the collected data mining to satisfy the following criteria:

- the Sprints contained in the time window must be temporally consecutive weeks
- the Scrum must satisfy the threshold such that the average of the commits of the first n-1st weeks must be greater than the following week n: $avg(n-1)^\circ > n^\circ$
- avoid Sprint overlap

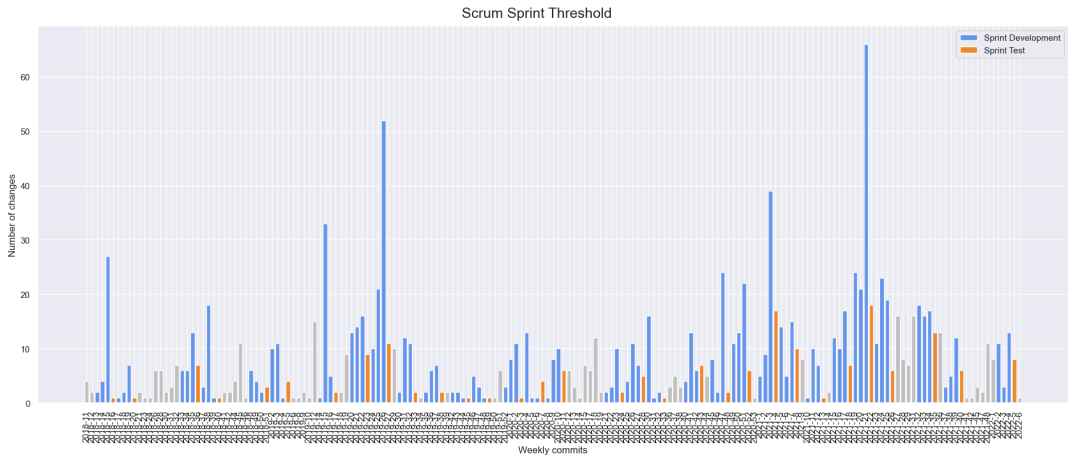

FIGURE 4.10: *Average Sprint Window* (ASW) on 'tag-security' (CNCF)

Figure 4.10 represents the result obtained on 'tag-security' project. It shows the activity score of weekly commits along the y-axis, while on the x-axis it shows the effort of each week when the project was under work. The bar chart highlights in light grey labour week that do not satisfy the filter criteria, in blue the consecutive weeks of development and in orange the testing one that suits the sliding windows set (4 in this use case).

What must be taken into greater consideration is how much frequently the window fits in the evolution of the repository. The more occurrences spread in the data, the more reliable will be the results obtained from the metric used. We can state that this project's activity is stable under the principles of Scrum framework.

The analysis carried out here is on the whole evolution of the development project with the purpose of checking and verifying the regularity of Sprints behind the repository to see if it has been constant over time and not reveal instead irregular manner. This makes us understand whether developers effort contributes in a continuous and regular way to the project, if the Sprint recurrence is respected or not and to identify possible faults.

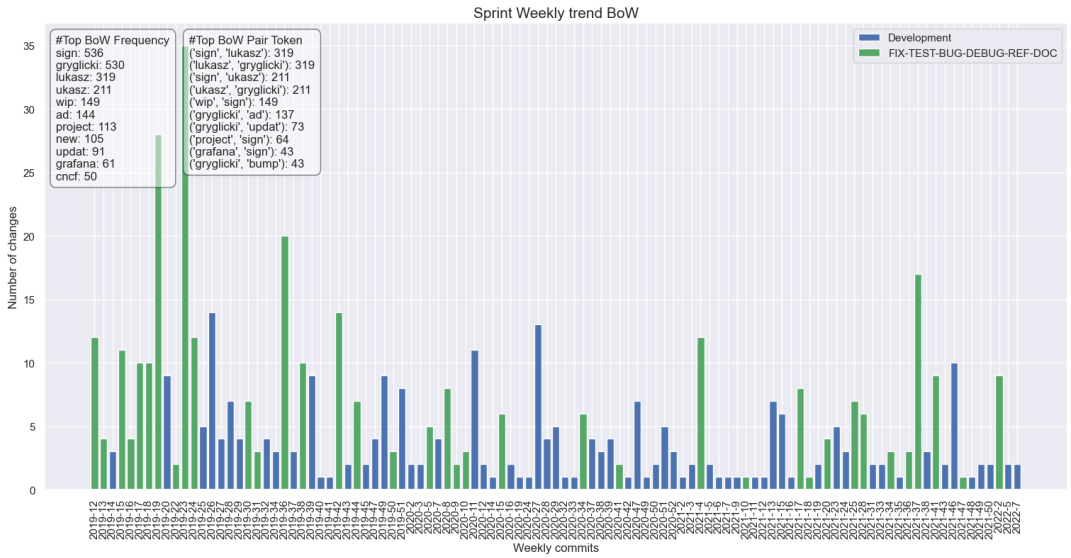

FIGURE 4.11: *Bag-of-Words Sprint commit message* (SCM) on 'devstats-docker-images' (CNCF)

4.6.3 Sprint commit message

The metric *Bag-of-Words Sprint commit message* (SCM) consists of identifying the possible presence of a Scrum Agile approach based on the study of a Bag-of-Word model on commit message. The measure is based on the recognition of Sprint time frame by studying the content of weekly commit messages. The BoW technique application, achieved through Natural Language Processing techniques to word normalisation (stemming), determines the presence of keywords such as FIX-BUG-DEBUG-DOC-REF-TEST to distinguish commits made during the testing phase from the development of the repository. Based on the presence of these keywords the Sprint week is tagged as developer or testing Sprint.

The script performs all the preprocessing steps of BoW. It consists of the stop word removal from the text of the messages, converting words to a common base domain with the operation of stemming and finally searching for matches with the words that concern us (FIX-BUG-DEBUG-TEST-REF-DOC). Depending on the keywords occurrences in the commit messages over the weeks, they associate a tag to the Sprint to which they belong. A collection process of stemming, stop word removal, tokenization, keyword matching is shown in Appendix A.2.

Figure 4.11 shows SCM for 'devstats-docker-images' CNCF repository. As we

can see, the data represent the Sprints tagged as testing weeks (in green) and tagged as development (in blue). The aim of this metric is to reveal the presence of testing Sprint, how much the fixing and debugging phases are distributed over the evolution of the project. It would be possible to recognise the testing weeks, how much effort the developers put in these Sprint, which Sprint test are most labours and if it was planned.

One more optional aspect that emerges from the representation is the possible presence of consecutive test Sprint. This came out when a single Sprint is not enough to manage all the old and new testing created for the occasion. Due to poorly performed testing results, fixing and debugging the next Sprint is extended by a week, giving enough time to solve any problems.

Considering that different developers teams use different ways to communicate and report a commit change, it could be possible to miss important and appropriate keywords. To better perform the BoW model, a legend shows the most commonly used words on commit messages. From this, it is possible to consider what type of keywords best characterise the testing phase. The keyword set (FIX-BUG-DEBUG-TEST-REF-DOC) can be edited accordingly to get a custom result.

4.7 Conclusions

Scrum is a team oriented agile framework that provides steps to manage and control the software and product development process. In this chapter, we formulated and implemented a couple of metrics to offer companies the opportunity to extract workflow behaviours from the development progress undertaken by Scrum team. We give the possibility to examine historic trends of repositories to evaluate the effort of developers. The metric outcomes identify the periods of increased traffic effort against the periods of lower, in order to distinguish and categorise the types of Sprints and delineate possible Sprint cycle windows.

Sprint week commit trend (SWCT), Average Sprint Window (ASW) and Sprint commit message (SCM) have been used to process mining to more than 113

CNCF repositories to detect the presence of Scrums framework on Open Source repositories (area difficult to analyse given the limited resources available).

The analysis can be performed both on the total repository history or on the individual remote branches that compose the project, to find possible branches dedicated to the testing phase and branches dedicated to the implementation of features, as well as to find which branches are most present and better characterise the repository history. It also included the opportunity to view the actual number of authors who participated to the software development to get an efforts overview of each individual team developer.

A method has been developed to detect time windows for sprints that report the average effort used in software implementation weeks, which are usually more labor-intensive than sprint testing weeks. A more in-depth analysis was conducted to determine which commits best characterised the development sprints to clearly distinguish development weeks from testing weeks.

To meet the demand to understand if the same development team is performing well over time, we focus on trying to identify which Scrum practices can be discovered using process mining techniques. We offer companies the opportunity to analyse the development progress undertaken by the team, to detect the improvement in effort estimation accuracy during the project life. Positive results are dictated by the evident presence of regular Sprint trends. With these pieces of information, the development teams will be able to take actions and decisions to improve their code and development process.

Chapter 5

Skill profiling in Software systems environments

The element that most affects the cost and management of a software project is the human factor. Understanding the elements that make up the team, their skills, and gaps would allow for the discovering of useful information, informing conclusions, and supporting decision-making. Being in a better position to understand what professional experiences are most relevant to developers would allow for the decision of the role best suited to developers, to understand the developer behavior in front of difficulty and, certainly better management of staff in software companies. One way to get this information is to study the tools used by developers and how they are used. Recent software advances have led to an expansion of the development and usage of application programming interfaces (APIs). Learning how to correctly and effectively use existing APIs is a common task and a key challenge in software development. Understanding the constraints and effort employed to use and apply APIs would allow for better profiling of developer skills.

5.1 Uncovering API usability

With the advent of new and more fields applied to IT, engineers and researchers try to provide useful and performing tools to reduce developers' programming effort and promote software quality. This is supported through Application

Programming Interfaces (APIs). Frameworks and libraries that provide access to implemented and well consolidated functionality, to the point of becoming indispensable for developers.

In a definition of an API pioneered by Martin Fowler, API is a "set of rules and specifications that a software program can follow to access and make use of the services and resources provided by one or more of its modules" [Red11]. An API is identified by a name and it consists of modules. Each module can have one or more source code packages of pre-made functionality [UKR21].

APIs have become an integral part of software development thanks to the increasing presence of standard and official APIs, ranging from the Java Software Development Kit, which comes with thousands of components that developers can reuse in their projects, to millions of open-source packages available in Maven, PyPI, and npm [LGS21; Rob09]. In response to shifting programmer needs and interests, libraries and APIs are increasing in number and complexity. The numerous open source APIs that are available for any given task are massive and constantly changing. Once an API is chosen from among the numerous APIs available, the developer must learn how to properly use it [UKR21].

An API usage is a piece of code that uses a given API to accomplish some task. It is a combination of basic program elements, such as method calls, exception handling, or arithmetic operations. The combination of such elements in an API usage is subject to constraints, which depend on the nature of the API. For example, two methods may need to be called in a specific order, division may not be used with a divisor of zero, and a file resource needs to be released along all execution paths. If a usage violates one or more such constraints, it is called a misuse [Ama+18].

Learning how to correctly and effectively use existing APIs is a common task and a key challenge in software development. It spans all expertise levels, from novices to professional software engineers, and all project types, from prototypes to production code. The importance of APIs is such that joining a company can require learning a whole new set of proprietary APIs before a developer becomes an effective contributor to the company [Gla+18].

Due to the constantly changing requirements of its users, the design and update of APIs are complex tasks. Some design decisions can influence the behavior of the API in subtle ways that confuse developers: useless information embedded, the information related to an API scattered in different parts, absence of example code or code examples without explanation, lack of focus in long text of features and requirements, ambiguous, vague or even incomplete explanations. The success and maintenance over time of an API is in the hands of API designers who have the difficult responsibility to make their APIs useful, accessible to its satisfied users and competitive with other pieces of software (highlighting the differences and qualities that characterise the API compared to the competition). Towards this goal, API designers need to provide development tools, documentation, and tutorials to use APIs. Final users must adapt to these API changes and new future releases [Aha+18; LGS21].

A code example only demonstrates API usage in one specific scenario. To adapt the API into programming context, programmers must traverse across API documents for needed information. To improve their API knowledge, often developers have no choice but to look for alternative documents and information sharing resources, increasing the time spent on consulting tutorials. Question answering sites, such as Stack Overflow[1] (SO), have become a popular place for discussing API issues. SO is a large online community where millions of developers ask and answer questions about their programming needs. Developers post questions in SO about their different technical topics, such as how to use APIs correctly under specific usage scenarios, selection, code reviews, conceptual questions and troubleshooting of APIs. These peculiarities make SO particularly effective for novices. Issues API posts are invaluable to API designers themselves, not only because they can help to learn more about the problem but also because they can facilitate learning the requirements of API users [Aha+18; UKR21; Aca+16].

Despite the vast amount of work on API-misuse detection, API misuses still exist in practice, as recent studies show. To our knowledge, in the current state

[1]Stack Overflow https://stackoverflow.com/

of art does not exist any estimation measurement systems of API misuse effort. We are the first to investigate this area by implementing a series of metrics to measure by degrees different levels of APIs issues use.

In this chapter we design a set of metrics to detect systematic user effort experience based on Java language. A scalable mining software frameworks was implemented to examine the code evolution between commits and releases in software repository. This would allow researchers to improve improper API use detectors by treating new fields and broadening the vision of research.

We propose approaches to extract the effort of API use made by developers from a software repository with a set of data mining metrics. The proposed metrics are: Hard API Effort, Near and Git Skill Analysis. Such metrics allow us to identify the contribution in software projects and consequently the developers expertise.

The remainder of our chapter is organised as follows. Section 5.2 presents the role of developers. Section 5.3 presents the concept and principles that lead to improper use of APIs. Related work is shown in Section 5.4. Section 5.5 introduces the methodology of our work. Section 5.6 presents the proposed metrics and the experiments performed with GitHub code snippets. Finally, in Section 5.7 we draw our conclusions.

5.2 The individual developer contribution

A focal point in the development of software projects, independent of the kind of project under consideration, is what concerns the effective contribution of members on a development team [GKS08].

We usually define a development team as the set of developers who have added, modified, or removed lines of code to a certain class of projects. In [Cat+19], it is aware that such a definition might lead to the approximation of the real composition of the development teams. In fact, a team member might not necessarily contribute to the development of the test code.

Certain useful metrics allow you to identify the contribution of the individual in the development of a project. In the past, the metric evaluation for assessing the individual's contribution was one mainly focused on counting the number of lines that the developer had put in the project(LOC); today a software developer is not just required to write a code, but also to co-operate with other colleagues and making development decisions. A developer, for example, is able to make a substantial number of code changes through the simple use of multiple tools specifically arranged in his favour [GKS08].

This change has become more evident with the emergence of Open Source Software through which anyone can collaborate and take part in a project already started, joining in for a long or short duration, without any restriction [MPR19; BBW13].

The main activity of a software developer is to write code, but his activity also involves constructive debate, exchange of ideas and anomalies resolution (bug fixing). As a result, the repository source code, any remote collaboration platform and issue tracking software are widely used as a data source to analyse and evaluate the level of participation and activities in a software project. First of all, it is useful to identify the resources of a project that can give a contribution and then to analyse their effectiveness. Not all actions, indeed, have a positive effect in the realisation of a project. For example, a commit on a repository can lead to a bug or a decrease in project quality, and therefore it should be measured by reference metrics and assigned with a negative weight [GKS08; Rob+14].

Unlike working on a team in a company project, where all team members well known their peculiar influence on project implementation, in an Open Source project this analysis is de facto complicated.

The opportunity to easily contribute to the implementation of a open source code without the need for an ad hoc procedure (simple code review on a merging request), on one hand they are positive factors of elasticity, but on the other hand they do not allow to evaluate the weight and effectiveness of the contribution of the individual to the realisation of the whole. It is possible for

a developer to push a single commit throughout the project and be considered equal with those who supported it for a long time [Rob+14; SMS16].

5.3 APIs should be easy to use and hard to misuse

Despite the efforts of its designers, APIs may suffer from a number of issues that can negatively impact the software developers' productivity by using APIs incorrectly. API errors are usually caused by stressful environment conditions, which may occur in forms such as high computation load, memory exhaustion, process related failures, network failures, file system failures, and slow system response [AX09]. Problems are often related to incomplete or erroneous official documentation, poor performance, system crashes and backward incompatibility of APIs [Nie+21; Wen+19].

As a result, developers often spend a long time trying to make existing APIs work for them (often imposing usage constraints, such as restrictions on call order or preconditions), and might end up writing code from scratch rather than using a difficult-to-use API [Sty+08].

All these may lead to introduce software bugs, crashes, data-loss, and security vulnerabilities. Software security is one aspect of programming where API utility makes a clear difference, but other domains like reliability and performance may also suffer from poor API usability [Mur+18; Gor+18].

Researchers have created several labor-intensive methods of uncovering API usability issues. Among the known techniques Murphy-Hill et al. report the following: user-centered design, heuristic evaluation, peer review, lab experiments, and interviews. However, only four are distinguished by researchers for their peculiar properties: an upscale approach that easily targets a large number of developers conducting surveys; examine API usability challenges by summarising them on expertise discussion forums and Q&A site (e.g. Stack Overflow); testing web API platforms to find any possible 404 errors that indicate which

APIs developers have problems with; and mining software repositories technique to look for instances of API on snippet code [Mur+18; Nie+21].

Existing API detectors commonly mine usage patterns (i.e., equivalent API usages that occur most frequently), and report any deviant code respect to these patterns, taken as the optimal model, as potential improper use. Our intent is not to rely on predetermined API patterns, but to study and have an even higher-level approach to detect possible mismatches in the use of an API over time.

5.3.1 Inappropriate use of APIs factors

The API can hide several issues it suffers from. Before a developer correctly uses an API, they may struggle by various learning barriers. This reveals the nature of the API usability problem [Mur+18].

The first key resource to learn APIs by the average developer is to study the corresponding official documentation. Programmers must manually traverse all API documents; a time-consuming and error-prone process to confirm the API usability [Gla+18]. However, it could not be exempt from mistakes. Official documentation is typically dominated by textual descriptions and explanations and and when it is no longer maintained, it can become incomplete, ambiguous, incorrect, outdated and even obsolete [UKR21; UR15]. Finding descriptive, non-ambiguous names for API features is problematic. Understand what each feature of the API represents and how it should be invoked, such as discovering relations between API elements require significant effort by developers. It should be guarantee that programmers can proficiently use the API without knowing (or assuming) implementation details and using the API it should be know how much positive impact on performance [Rob09].

Developers complain about the lacking concrete code examples that illustrate API usage. A lot of API reference documentation lacks code samples with usage scenarios. Otherwise a simplified code example often excludes the possibility of alternative API uses. Aspects that programmers frequently desire when learning unfamiliar APIs [Gla+18]. According to J. Zhang et al.'s statistics, only 11 and

6 percent of API types are illustrated by code samples but with vague usage scenarios in the Java Standard Edition (SE) (version 7, 2019) and Android API documentation (version 4.4, 2018), respectively [Zha+19].

Possible further gaps are not exclusive to documentation error. By mining bug repositories, it is possible to detect mainly the presence of human error issues. Papers reveal API usability challenges including simple typos, conceptual API misalignments, and conflation of similar APIs. Change call mainly due to a similarity of the API invocations or dubious knowledge of what different outcome these methods may have [Mur+18].

An underlying problem of API usability issues can be related to an initial mismanagement design. The task of designers is to avoid these cases that lead to incorrect behaviour in applications. Faults due to poor memory handling, breaking changes that lead to backward incompatibility, conflict of the APIs with underlying operating systems or inconsistency interaction to other external libraries [Aha+18].

Not least important aspect to consider is the developer API experience. Developers who have not used API before are more likely to reflect actual API usability problems, compared to developers who are already familiar with the libraries [Mur+18].

5.4 Related Work

In prior research have been proposed several API detectors and various representations of API usage patterns and a multitude of automated bug-detection approaches. These detectors analyse code snippets that use a given API. The detectors commonly mine usage patterns from source code and find rare violations of those. They report deviations from frequent patterns as potential wrong use, assuming that they often correspond to bugs.

The approaches differ in how they encode usages, as well as in the techniques they apply to identify patterns and violations. In general, the procedure of mining usage patterns covers the following five steps [Nie+21]:

- Collect a representative set of source code for mining.
- Transform code set into an intermediate representation, e.g., execution traces, syntax trees, API usage graphs and so on.
- Conduct a frequent pattern mining approach on this representation.
- Filter generated patterns based on suitable ranking metrics.
- Compare and report violations as misuses.

An example is made by Amann S. et al. who make use of the term *API misuse* as violation of constraints of the API. To mitigate weaknesses of existing detectors they design MUDETECT, a graph-based representation of API usages. Nodes represent data entities (variables and method calls) and edges represent data flow and actions amid nodes. The graph captures many properties that can distinguish misuses from correct usages. A detection algorithm uses domain knowledge to efficiently identify pattern violations and, to improve precision, a ranking system to avoid false positive results [Sve+19].

The challenge to determine and understand if an API misuse is real (erroneous call and not unwanted but functional one) are usually overcome by using three independent approaches, as explained below.

Analysing, exploit, explore and expanding existing bug datasets. A manual or heuristically approach to filtering instances of arbitrary software bugs, starting from data made available by the contribution of previous research. Several datasets of software bugs have been created in the past [Ama+18; JJE14; Fer+20a]. Many researchers created bug datasets to evaluate their approaches and make them available to contribute to the benchmark detectors comparability. Also such datasets are useful as input for classification models and forms a good base to support replications and comparisons by other researchers.

The second approach is understanding the nature of the bugs and bug-fixing processes associated with open-source projects. Projects on desktop, server applications and smartphone platform are some of the most popular types of projects software mined among the more established open platforms such as

SourceForge and GitHub [Bha+13; Tan+14; HE22]. Repositories vary by type, programming language and context. Today large open source developments are burdened by the rate at which new bug reports appear in bug repository. This is a great starting point to get more information about it. Actually different techniques are applied on Machine Learning-based Techniques to the open bug repository to learn the kinds of reports each developer resolves and detect bug severity (a different levels of attribute degree based on the bug impact on the system) [AHM06; CS12; Thu+12].

A further common approach is to conduct literature reviews, surveys and interviews asking developers about the obstacles they faced learning APIs. It is well known that during development and through testing many hitches are already ruled out from the repository. Thus, developers might face many misuses that we cannot find by reviewing repository [Ama+16]. The surveys aims are uncover many challenges to the development, usage, discover learning strategies, and evolution of APIs. The questions often cover the critical aspects of API usability with a view to get a detailed picture of important obstacles developers faced when learning new APIs. While the purpose of the literature reviews is to study the progress and trend over time of research and paper in the field of usability and maintenance of APIs and understand in which direction it is evolving [LGS21; Won+16; Rob09; PFM13].

Nielebock et. al. also faced the same problem. They focus on API usage patterns that are inferred from existing source code through data mining. An API change analysis phase is performed with each API call change. To avoid inadequate collection of source code samples from which to infer patterns which leads to false positive results, they rely on a search and filter process of source code files with similar but correct API usages using different strategies. Taking advantage of data representation under structure like graphs or trees the miner conducts a frequent pattern mining approach to generate a list of ranked API usage patterns. From these results is possible to generate patches and validation controls under the API misuse detection for possible end fixing commits [Nie+21].

Jadeite is the prototype made by Stylos et. al. A tool system that gives the possibility to mine API from Google, code repositories, or explicitly annotations by programmers to improve existing API documentation. Their main property is adding useful information in a controlled way. The motivation for this feature comes from observing programmers become frustrated with APIs that did not contain the expected classes and methods. The system provides the possibility to programmers to add placeholders for operations that the original designers never thought about or show code explaining on how to accomplish the desired functionality with the available APIs. So that programmers do not need to re-learn the API when returning to it in the future because they get a spot on which find these informations [Sty+09].

There have been many attempts for improving the usability of existing APIs an example is changing the integrated development environment (IDE). Ideally, IDE should assist developers in implementing correct usages and in finding and fixing existing misuses. And that's what Wu et al. implemented; an Eclipse plugin CoDocent to help programmers confirm the semantics of the API calls by various code search engines. CoDocent summarises all API classes traversed and organises these classes according to the package structure in diagrams. CoDocent can be effectively used in selecting and investigating relevant code examples for programming with APIs [WMJ10].

The popularity and growing influence of SO has motivated a number of recent research efforts to produce API documentation automatically from SO contents, such as adding code examples and interesting textual material by summarising API reviews (with positive and negative sentiments to assist API selection) [UKR21]. While a lot of research has focused on finding code examples for APIs, Treude et al. present an approach to improving or augmenting the descriptions contained in API documentation with "insight sentences" from SO. Their techniques assign a numeric value to each sentence and return the top-ranked sentences of SO posts (question, answer, authors, etc.). A developed machine learning approach called SISE (Supervised Insight Sentence Extractor)

import the insight sentences which uses as features to understand possible differences present or missing important aspect in the documentation to explore further [TR16].

Customised code search engines are becoming increasingly popular. They are designed to index all information of software projects e.g., help documentation and textual descriptions of applications. Moreover, general purpose search engines are not designed to receive snippets as queries. Typically the reason is that these engines do not work well with large queries. Campos et al. try to mitigate this problem. They propose an approach to find fixes for API-usage related bugs, which is based on matching code snippets of SO written in Java and JavaScript programming languages. The aim is to recommend posts of SO whose code match developers' snippets. To achieve this they define some preprocessing functions for code snippets: removing all punctuation characters, generating an abstract syntax tree for the code and extracting their API method calls. Only once collected these data they perform experiments on a relational database which represents a release of SO public data containing also all answers to each question, if any [MCM16].

Wen et al. propose the first automatic approach to discover API misuse patterns via mutation analysis. While mutation analysis has several application domains on testing, fault localisation and security; the authors' observation was born from the fact that API misuse can be viewed as the mutants of API's correct usages. Therefore, API misuses can be created via applying specifically designed mutation operators on correct API usages. To progress in their analysis, they design eight mutation operators that mimic different kinds of programming mistakes. A benefit of this approach is that it does not require pattern mining from a large number of correct API usages. Instead, it actively creates substantial mutants mimicking API misuses of different patterns [Wen+19].

5.5 Approach

Finding and initially studying an API is the first and one of the most common steps in using APIs. APIs are so large that people often need to learn to use various aspects. It would be a misfortune if the difficulty of using APIs would nullify the productivity gains they offer.

Since it is difficult to obtain sufficient correct API usage examples in practice, especially for newly released third-party libraries. To improve precision, upcoming detectors need to go beyond the oversimplified assumption that a deviation from frequent usage patterns is a misuse. An uncommon usage of an API is not necessarily an incorrect usage [Ama+18].

Software fault localisation is widely recognised as one of the most tedious, time consuming, and expensive activities in program debugging. Due to the increasing scale and complexity of software today, manually locating faults when failures occur is rapidly becoming infeasible, and consequently, there is a strong demand for techniques that can guide software developers to the locations of faults in a program with minimal human intervention [Won+16].

For this reason, while most previous research has focused on studying the usability of specific APIs. Unlike proper error and exception handlers support for managing API errors, our research seek to obtain more generic results by studying APIs misuses patterns that occur in the most varied possible APIs. We aim to exploit the common characteristic of misuses that occur repeatedly in different contexts, independent of the application domain.

We focus on those points and aspects that make APIs difficult for programmers to use. We define several metrics to highlight the quality of misuses and analyse the developer involvement in the bug-fix process and how they structure their code accordingly. The purpose of our research is to provide additional data that can aid in the evaluation of API effort estimation.

We have implemented an analyser to determine how efficiently APIs are used by providing a targeted usability evaluation. All metrics are deployed on PyDriller, a Python framework for MSR [SAB18] capable of mining arbitrary Git

repositories, to extract all core Git data, such as commits, source code modifications, timestamps, etc. We offer a tool that can be integrated with the analysis system already established to expand the study and monitoring of developers' API efforts.

To study the feasibility of the metrics we identified the most forks and rating stars Java open source projects hosted on GitHub. We randomly selected 34 popular projects on which to perform analysis of the associated metrics tests. The list of projects and their characteristics are shown in Table 5.1. It shows the name of the repositories, the number of commits present, the number of contributors, the associated star rating and the percentage of .java files present. We do not know the identity of the organisation that developed these projects; however we are aware that the projects were developed for a variety of industry sectors. All data thus collected is made available in a dataset to anyone who wants to extend the research.

To meet the demand to understand if the development team is performing well over time, we offer companies the opportunity to analyse the development effort undertaken by the team. We provide the possibility to examine with a single global view the detected and trending changes of API call usage in the repository. Project Manager and the development team can use the proposed metrics to monitor the progress and improve the accuracy of programmer effort estimation during the project life cycle.

5.6 Metrics

To advance the state of the art of API-misuse detection, this section describes the proposed metrics and the experiments performed on open source repositories.

We defined a few processing functions for code snippets to detect API calls and their progressive use change, without requiring any further user input (such as: setting a target API to study, programmer observations, the maximum number of API calls to analyse, etc.). In this manner analysis is done on all

Projects	Commits	Contributors	Stars	Java
Android-Universal-Image-Loader	1.038	38	16.800	100%
gson	1.727	132	21.400	100%
fresco	3.335	217	16.868	78.4%
APIJSON	2.801	47	13.828	100%
elasticsearch-analysis-ik	259	35	14.352	100%
AndroidUtilCode	1.417	38	31.422	88.3%
HikariCP	2.822	124	17.340	98.4%
zxing	3.674	114	30.181	96.2%
Apktool	1.893	87	14.915	98.5%
incubator-streampark	914	74	2.507	34.1%
termux-app	1.411	61	16.232	98.2%
xManager-Spotify	589	14	2.759	100%
jsoup	1.710	100	9.758	81.6%
doris	6.684	393	18.356	46.4%
PhotoView	462	40	6.212	100%
besu	3.772	138	997	99.6%
ysoserial	167	29	5.764	99.8%
Infinity-For-Reddit	1.868	36	2.307	100%
BaseRecyclerViewAdapterHelper	1.231	37	23.149	55.7%
canal	1.533	174	24.244	94.3%
arthas	1.809	164	30.739	65.5%
sentinel	764	170	20.005	89.9%
CircleImageView	159	14	14.317	100%
easyexcel	874	44	25.396	100%
Material-Animations	76	9	13.565	100%
logger	146	11	13.431	62.5%
JustAuth	692	29	13.853	99.6%
butterknife	1.016	97	25.674	94.0%
ARouter	301	22	13.972	90.0%
seata	1.537	250	22.910	97.0%
nacos	4.266	259	24.262	98.8%
SmartTubeNext	6.258	54	7.460	98.3%
NewPipe	10.254	740	21.418	83.7%
Mindustry	15.694	488	16.232	99.3%

Table 5.1: List of top stars rating Java projects on GitHub

API calls present in the repository. Similarly, knowing which APIs and libraries are imported, it is also possible to generate a developer profiling system.

Given a URL of a Java GitHub project, the framework analyses the modified .java files in each commit of the repository. A dataset was made for each data analysis in order to report the computed results and their degree of usefulness. Each entry will contain specific information such as: API type, instance name, method invoked, line code and more. By analysing these data, API application and effort involved can be estimated, allowing to report a pattern of the behavior and actions of the development team.

5.6.1 Hard API Effort

The implemented metric *Hard API Effort* (HAE) attempts to detect how complex it is to use Java libraries. To define the effort of API use, for each modified source lines of code the implemented script focus on:

- the number of changes related to objects instantiated from classes constructor;
- the number of changes of method invocations from class or API library;
- what kinds of methods are most frequently cited;
- which APIs have given more trouble to use.

The first step consists in sorting the URLs repositories links to be processed, by filtering out inappropriate URLs and removing possible duplicates. An in-depth analysis is conducted on each specified repository. In each project commit only the modified Java files are analysed. A file is considered to be modified if it has undergone a change (MOD), a code line addition (ADD) or a code line removal (DEL).

To investigate the use of APIs over time throughout the evolution of repository development, in each commit a research is conducted to detect the presence and type of classes and APIs used. In this way, it is possible to see which APIs have been used for the most part and which ones have been disused by the

Filename	Line	Change type	Code
AbstractBigQueueTest.java	65	['ADD']	[List < BytesValue > dequeued = new ArrayList <> ()]
AbstractBigQueueTest.java	75	['ADD']	[dequeueingFinished.countDown()]
Account.java	25	['ADD']	[return new Account(name, KeyPair.generate()]
Account.java	34	['ADD']	[keyPair.getPrivateKey().toString()]
Account.java	42	['ADD']	[return BigInteger.valueOf(nonce++)]
...	...	...	...

TABLE 5.2: Extract from data frame Tokens of 'besu' project: input

development team, possibly due to a difficulty to use factor or because of lack of aimed developers requests and expectations.

For each edited code line the script filters out any single and multiple-line comments. A function processes the Java code snippet and extracts their API method calls and instantiations using regular expressions. A regex processes the code snippet and identifies all object declarations and all API method calls occurring in the code. The main desired effect is to remove any extra code lines that are not of our interest.

Once this is complete, the tool parses the Java code by splitting the line in pairs of values: tokens and corresponding category value (Identifier, Operator, Separator, Method, Function, Class, Variable, etc.). An example of HAE application is reported in the following tables on 'besu' repository, an Ethereum client written in Java by Hyperledger Foundation organisation. The intent is to detect the effort employed by the team in using the applied APIs and to detect which APIs were more difficult to manage.

Table 5.2 shows the first input values of the first lines of code of the initial commit of the project. You can see how each entry is characterised by the name of the current java class, the change type, index of line of code and the code snippet. Through this table it is possible to monitor any possible changes to the API calls present in the project, how many times they have been modified and what type of modification has been made. Table 5.3 shows the data frame outcome of code so divided into tokens values. By breaking down the code into tokens, it is possible to trace back to the variables and their API calls as reported in the following tables.

Tokens
[(List, Class), (<, Operator), (BytesValue, Class), (>, Operator), (dequeued, Variable), (=, Operator), (new, Keyword)...
[(dequeueingFinished, Variable), (., Separator), (countDown, Method), ((, Separator), (), Separator), (;, Separator)
[(return, Keyword), (new, Keyword), (Account, Function), ((, Separator), (name, Variable), (,, Separator), (KeyPair, Class)...
[(keyPair, Variable), (., Separator), (getPrivateKey, Method), ((, Separator), (), Separator), (., Separator), (toString, Method)...
[(return, Keyword), (BigInteger, Class), (., Separator), (valueOf, Method), ((, Separator), (nonce, Variable)...
...

TABLE 5.3: Extract from data frame Tokens of 'besu' project: output

Filename	Var-name	Var-type
AbstractAltBnPrecompiledContract.java	errorString	String
AbstractBLS12PrecompiledContract.java	errorMessage	String
AbstractBigQueueTest.java	dequeued	List
AbstractBigQueueTest.java	dequeueingFinished	CountDownLatch
AbstractBigQueueTest.java	queuingFinished	CountDownLatch
AbstractBlockCreator.java	isCancelled	AtomicBoolean
AbstractBlockProcessor.java	blockHashLookup	BlockHashLookup
...	...	...

TABLE 5.4: Extract from data frame Variables of 'besu' project

The importance of these data frame is given by the support they can provide. Table 5.4 represents the data frame Variables, which keeps track of the instances present in the project and the associated Class. Several checks are carried out on object instances to monitor the used methods and to answer the question to which Classes they belong to. The intention is to detect which are the main instances and related Classes that are mainly used in the project and consequently, to have an overview of which are the main libraries and APIs used in the project.

Instead Table 5.5 shows data frame Methods, which represents the methods detected by the metric. The data frame catalogs the methods invoked, their Class of affiliation, and the respective line and class in which they have been invoked.

Once the mining and parsing operation is complete, the script counts the number of times each method appears in the changed lines set. The count of modifications and the list of classes where the invocation is made are information extrapolated from previous support data frames. The result is shown in Table 5.6 below. Each entry reports, sorted by most widely used methods, the method name, the class which it belong, the number of times the method was

Filename	MethodName	Class	CallingClass	Line
JsonRpc.java	waitFor	WaitUtils	JsonRpc	17
WaitUtils.java	await	Awaitility	WaitUtils	10
Account.java	generate	KeyPair	Account	25
Account.java	create	KeyPair	Account	29
Account.java	create	PrivateKey	Account	29
Account.java	fromHexString	Bytes32	Account	29
Account.java	extract	Address	Account	38
Account.java	hash	Hash	Account	38
Account.java	valueOf	BigInteger	Account	42
...	...	...	...	...

Table 5.5: Extract from data frame Methods of 'besu' project

MethodName	Class	Count	CallingClasses
of	Optional	4883	['PantheonNode', 'LoadedBlockHeader', 'CliqueVoteTallyUpdaterTest'...
add	List	2255	['Accounts', 'Cluster', 'ProcessPantheonNodeRunner', 'WebSocketConnection'...
of	BytesValue	1526	['PantheonNode', 'LoadedBlockHeader', 'CliqueVoteTallyUpdaterTest',...
put	Map	1469	['Cluster', 'ProcessPantheonNodeRunner', 'ThreadPantheonNodeRunner',...
getLogger	LogManager	1450	['Cluster', 'ProcessPantheonNodeRunner', 'ThreadPantheonNodeRunner'...
newArrayList	Lists	1338	['CliqueExtraData', 'CliqueDifficultyCalculatorTest', 'CliqueBlockCreatorTest'...
info	LOG	1240	['PantheonNode', 'ProcessPantheonNodeRunner', 'ThreadPantheonNodeRunner'...
toList	Collectors	1218	['PantheonNode', 'ProcessPantheonNodeRunner', 'IbftBlockHashing'...
emptyList	Collections	1216	['VoteTallyCacheTest', 'IbftBlockImporterTest', 'BlockAddedEvent'...
fromHexString	Address	1146	['Account', 'LoadedBlockHeader', 'CliqueExtraDataTest'...
fromHexString	BytesValue	1079	['Account', 'LoadedBlockHeader', 'CliqueExtraDataTest'...
...	...	...	...

Table 5.6: Extract from data frame HAE count of 'besu' project

modified, invoked and deleted throughout the project history and list of classes it was called. Through this it is possible to derive which API libraries have been more difficult for developers based on the changes that have been applied in the development of the project and which consequently required greater effort from developers.

Possible methods and instances not included are listed in the generated log files. This happens when the statements of these instances are not available; for example object passed as parameters, casting conversion or invocation of imported library methods (e.g. System.out.print).

APICall	ConsecutiveModifyLine	APIModify	CorrelationModify
[final int inputSize = Math.min(inputLen, input.size());]	0	False	7
[messageFrame.setRevertReason(Bytes.wrap(error, 0, err_len.getValue()));]	3	True	8
[LOG.info("Native alt bn128 not available");]	0	False	1
[return Bytes.wrap(result, 0, o_len.getValue());]	5	True	2
[int inputSize = Math.min(inputLen, input.size());]	0	False	0
...	...	...	...

TABLE 5.7: Extract from data frame *Near* of 'besu' project

5.6.2 Near

Most of the fixing commits resolve multiple misuses. Commits contain more changes than the fix itself, such as refactoring or reformatting, which makes monitoring the actual misuse fix more difficult to detect. To address this challenge, the metric *Near* is designed to provide evidence for misuse APIs based on the presence of what surrounds a method call.

The metric tries to detect how the difficulty of using API calls is possibly related to different factors. On the basis of several experiments, it was concluded to divide the metric into three different assets in order to provide three levels of granularity dictated by their distinct combinations. The measurement is a combination of balanced assets to distinguish and approximate different API use context in order to achieve a classification of API misuses.

The metric consists of: identify lines changed before an API call, detecting changes to the parameters used by the API call and report any modification on the API call itself. Based on the presence of these, it is possible to weigh differently the result of the metric and therefore the amount of effort which was required for its appropriate use.

Similarly to Hard API Effort metric, the metric *Near* has the same initial examination phase. Once the range of lines to be analysed before an API call is established, the data mining script leads to obtaining data frames containing the expected results. An outcome of 'besu' project is shown in Table 5.7. The Table gives an extract of the outcome of *Near* on the AbstractAltBnPrecompiledContract.java file. The file name, the commit hash and the commit date have been omitted to render the Table easier to read.

For each method invocation the following values are considered:

- *ConsecutiveModifyLine* represents the lines consecutively changed before the API call. If the lines just before an API invocation were changed, it could be a change that will affect the next API call. Once the size of the sliding window row is set (five by default) what is calculated is the consecutive count of lines modified before each invocation of API method present in the code.

- *APIModify* signals the modified API call event. Whether present or not, it implies an important role in evaluating the effort of using the API.

- *CorrelationModify* represents how many mentions to the API call there are among the changes made before an API call in the same commit. *Near* detects any reference and occurrences related to the object variable name and parameters used, as input, by the API method that is present in the changes commits history.

In conclusion, it should be noted that the mining of first commits can report false positive outcomes. Each line of the new files inserted in the repository is interpreted in git as a change. For this reason *CorrelationModify* reports as many changes as the size of the established window. The solution approach consists of skipping these first commits and analysing the next ones directly (starting from the presence of the file in the project and its subsequent changes).

5.6.3 Git Skill Analysis

Software project estimation involves the judgment of different aspects that have a significant influence on the creation of the final software product. Team skills, prior experience, and task size are cited as the three important cost drivers for effort estimation in ASD [AAA17; Usm+14]. It is deduced that the greater part of cost management is attributed to human resources. Therefore, estimating development effort is central costs in software development and the most difficult parameter to estimate [ARG06].

Due to the increasing importance of Open Source software, modern software

development processes involve multiple developers and development teams residing on different continents and in different time-zones [PSV11]. Looking for Open Source projects is better because it shows that there is a community scattered around the world and therefore we can approach certain contributions' profiles (from the most experienced person to the least).

We may know from the very beginning of the project that low capabilities of the development team will surely have a significant negative impact on project effort. The project manager identified potentially insufficient domain experience of the development team as a risk to project success. Team's domain experience is a relevant effort driver in the effort estimation process. After quantifying actual domain experience of team members and estimating development effort, it occurs that effort exceeds the customer's acceptable level.

The aim of the metric is to classify the committers on a given GitHub repository. The need derives from the disparity in team roles present in a project, including supporters and repository maintainers.

The developers' skills are highlighted by categorising the work into three macro categories: *backend*, *frontend* and *writer*. The programming language, libraries and file extensions used in the repository are taken as reference to analyse the project: e.g. *Writer*: pdf, txt, md, etc.; *Frontend*: css, php, html, etc.; *Backend*: sh, c, py, etc. There is an additional *undefined* category that stands for extensions supposed to be present in the repository but that you want to exclude from the analysis.

The program distinguishes which category to associate and tag with the respective libraries and APIs. In addition to these macro categories, there is the possibility to include additional extra categories by specifying their scope and associable libraries that you expect to find: e.g. android, facebook, etc. The final results can be saved in a csv or html format. All these repository information are contained in the support file *Config.properties* 5.1. All form fields can be customised as required for further analysis. Some fields are mandatory and necessary for the execution of the script (such as the HTTP address of

```
[RepositorySection]
repository=https://github.com/cncf/cnf-testbed.git

[SkillsSection]
backend=sh;py;c;cpp;go
frontend=css;scss;html;ts;ui;kt
writer=pdf;md;txt;tex
undefined=php;java;js
java_fe:javax.swing;java.awt;com.lowagie;org.xml
android:android.app;android.content;android.net;androidx.annotation;android.database;
android.support;android.os
facebook:com.facebook

[OutputSection]
export_as:html
```

FIGURE 5.1: Sample of *Config.properties* format

the repository to be analysed or the global path of the cloned repo, export file format), while others allow a more accurate analysis.

As the first step, the metric searches for all repository commit authors. The developer list serves to filter possible duplicate accounts. Since there can be commits made by the same user (user.email) with different aliases (user.name). Once the different authors who have contributed to the project are obtained, the metric determines a trend effort among all developers.

To define the effort of developers, each committer is assigned scores for each category. The scores are then converted into a percentage of the total work done into the repository. The score corresponds to the number of modifications made in the commit history that satisfy the requests specified in the config file. Specifically, once the commit author is established, the number of changes made to the code lines is counted.

The count is made by verifying the typology of modified files and increasing the number of lines that have been altered in the file in the appropriate category (backend, frontend, writer, undefined). According to the imported and used API libraries, you can measure the capability and skills of developers.

For the management of Java and JavaScript packages, we rely on npm[2] site package manager which, by specifying the name of the package, will give all the useful information. To read the site and get information on the different

[2]npm https://www.npmjs.com/package/

packages we rely on an html parser. By reading the "readme" site section, if it contains the word node.js, the package in question is treated as a backend; otherwise, it is treated as a frontend.

Developers' bio information is obtained through common REST API web services requests. Most Git repository hosting service support it. The script constructs an HTTP request, which goes to the web server and a response comes back. There are different replies formats, but JSON is pretty common use.

The developers' information thus collected is saved, the Table 5.8 shows the types of achievable entries in the csv format. Not all information is visible at a REST API request from a user outside the project: the information must be made available by developer profiles and only authenticated users can access the REST API query. However, the owners of the repositories can still carry out and obtain the sought information.

The result in html format is shown in Figure 5.2. It shows some of the commiters who collaborated on the 'cncf-testbed' project (Cloud-native Network Function Testbed, a set of reference code and test cases for running networking code on Kubernetes and OpenStack to evaluate CNF architectures). The intention is to recognise the expertise and profile the developers who have contributed and participated in the project, to provide the possibility of detecting and collecting the skills present in the team and finally to detect how and how much each developer has contributed to the project.

In the html version, everything is saved in a zip file containing a JSON file (with the same information obtainable in the csv format) and other elements for a web representation of the results obtained. For each developer, the following information is also retrieved: name, profile image, a star rating to indicate their contribution to the repository based on the total number of commits made, the percentage of effort per specified category and field of contribution, email, personal web site or blog and nationality. This can help companies organise the development team more efficiently by assigning tasks based on the skills of each team member. In this way, companies can maximise the efficiency and

Name	Name and surname of the developer
Email	Email
SocialID	Unique developer code on the git host
SocialUsername	Developer username on git host
AvatarURL	URL avatar set on git host
WebSite	Personal website or blog
Location	Nationality
Bio	Bio info
CreatedAt	Account date created on the git host
Commits	Number of commits processed
Backend%	Backend category percentage
Writer%	Writer category percentage
Frontend%	Frontend category percentage
CatExtra%	Percentage of extra category

Table 5.8: Skill Analysis csv output structure

productivity of the development team and improve the quality of the software project.

5.7 Conclusions

APIs are becoming indispensable for developers. Learning how to correctly and effectively use existing APIs is a common task. However, APIs may suffer from a number of problems that can negatively impact the software developers' productivity. On one hand, APIs reduce programming effort and promote software quality, but on the other hand, their increasing complexity makes it more difficult for end users to learn and use them.

In this chapter, we presented an approach to uncover effort API usability. Our metric helps to understand how much effort is needed to learn and master an API. The metrics measure different levels degree of API misuse. The aim of the metrics is to discover the developers API expertise, as well as the amount of effort employed in their daily use especially with unfamiliar APIs.

It is also possible given a GitHub username, to research the skills that characterise it. The result is obtained through a detailed analysis of all the APIs used in the project's history. This allows for the identification of actual roles and contributors based on the efforts that have been applied to the repository.

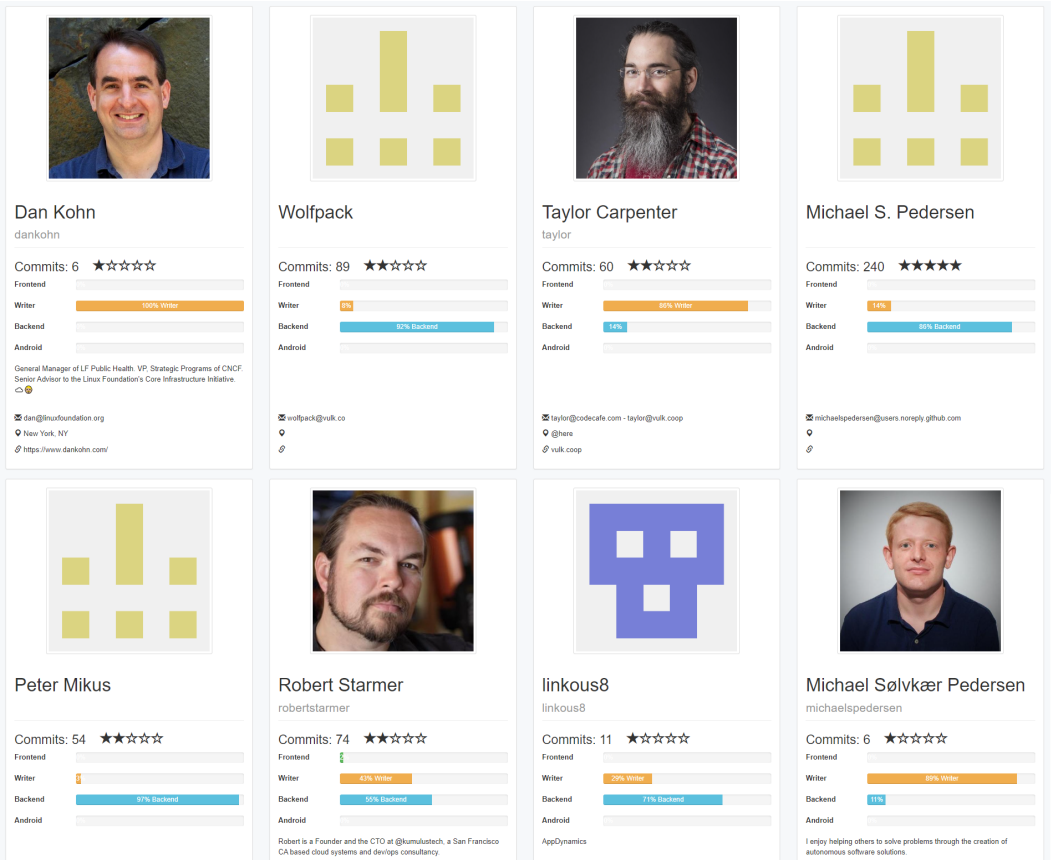

FIGURE 5.2: Skill Analysis html on 'cncf-testbed' (CNCF)

As current research focuses on the creation and perfection of automated misuse-detection tools, such as error and exception handling support, we hope that managers and researchers will use these observations to better profile members of development teams. We also hope that our series of studies will inspire others to study even more aspects of the usability of APIs, determining whether or not an API is used further, so that usability can be an important consideration for all future designs.

Chapter 6

Conclusions

This thesis has presented a support to assist effort estimation on software development, showing how data mining on descriptive information and metadata contained in the repositories can give useful insights to measure the commitment of the developers behind the creation of the progress. These results also indicate that data mining techniques can make a valuable contribution to the set of software effort estimation techniques.

Software effort estimation is one of the base activities of software project planning and accurate estimations are essential for successful project management. Many factors have an impact on the software development process as many are the risks brought by insufficient information, which can not be known in advance. In the literature, there are several known techniques and functional approaches, but no less important are the principles metrics at the heart of all models. Appropriate metrics can show their usefulness and importance to better draw and validate the results obtained by software effort estimation practices, as well as to monitor and refine the way to outline the progressive software development process. The aim of this thesis is to identify metrics, and according to such metrics evaluate the similarities among projects, to understand the relationship between the number of developers on a team project and the effort contribution of each developer.

A data mining tool has been implemented in the study of software repositories with the use of PyDriller, a python framework to analyse Git projects. PyDriller is a tool that take a definition of a theoretical metric deduced by the experts

in the area of mining software repository, and computes it according to the developer's effort found in the analysed project repository. Each of these metrics has its own precise meaning and purpose. PyDriller allows you to perform an analysis and draw the necessary considerations and gender statistics on various projects even large scale ones. Each metric used on this tool determines results that can be used to draw relevant conclusions.

The results and outcomes obtained in this thesis derive from an experimental stage and analysis carried out on a hundred projects found on GitHub. This work created a sample of projects chosen appropriately and different from each other, in order to better characterise the various results collected, based on large Open Source projects. Open Source shows that there is a community behind it spread around the world and that, as a result, different profiles of contributions are present.

The study of commits was the central focus of the thesis. Metrics on the study of commits made it possible to understand the progress of the releases made and stages trend of the development project. It was demonstrated how to determine the frequency of the commits and in which time frame they were performed; it is possible to highlight: the periods of increased activity, in which time interval the work is mostly involved, or the periods of regular average activity trend, and how long the overall development lasts (*Average Commit Distribution*).

The analysis was conducted for: separate weekly day (*Commits per Day of the Week*) in order to highlight the different working days and compare weekly days from weekend days, separate hour of the day (*Commits per Hour of the Day*) to determine the usual working hours of the development team, separate commit per week to give a global project vision to highlight which periods of the year with slight developer activity (*Commits per Week in last Year* and *Changes per Week Trend*). As a side effect, by means of the collected metrics outcomes, trying to evaluate and find the differences that characterise them, it was possible to index three categories of projects such as Open Source, Side Project and Full Time Project.

To frame the work on specific characteristics of each project, an analysis was also

performed on good practices of agile development methods and processes. Special attention was given to the effort estimation that characterise the developer activity in Scrum agile framework, one of the most common agile methodologies. The utility of the proposed metrics was to reveal the activities performed during the developing of the project, the effort trend of developers, to detect Sprint week and possibly suggest improvements in the adopted Scrum practices.

An automatism has been laid down to give the possibility to examine historic commit trends to evaluate the effort of developers. A sliding window method was implemented to meet fixed asset values to detect consecutive development and testing sprints (*Average Sprint Window*). Positive results were dictated by the evident presence of regular Sprint trends.

Another important property addressed in this thesis was Bag-of-Word model application on commit message. The measure was based on the recognition of type of Sprint by studying the content of weekly commit messages. This Natural Language Processing technique allowed us to distinguish commit created during the fixing, testing, refactoring and documentation phase (*Sprint Commit Message*).

Measuring the effort of a developer is also related to measuring his skill and approach in using common available tools. Metrics have been proposed to reveal effort of developers and their practices on API misuse (*Hard API Effort*). If on the one hand, APIs reduce programming effort and promote software quality, on the other hand they could have an increasing complexity. Metrics have been presented to uncover effort API usability, measure effort with different levels of capability and profile the developer skills based on the types of API libraries usually used (*Git Skill Analysis*).

In conclusion, different aspects related to software development effort have been addressed by proposing metrics and analysing different metric applications. With such analysis researchers can empirically investigate, understand and discover useful information for software engineering. Project managers can benefit greatly from the metrics adopted as they can monitor the effort of development teams and support future project decision-making.

Appendix A

Listings

A.1 Average Sprint Window code implementation

```
from pydriller import Repository
from pydriller import Git
from git import Repo
[...]

logger = logging.getLogger(__name__)

def log(verbos):
    logger.setLevel(logging.INFO)
    formatter = logging.Formatter('%(asctime)s:%(levelname)s:%(name)s:%(message)s', datefmt='%d/%m/%Y %H
     :%M:%S')
    if verbos:
        stream_handler = logging.StreamHandler()
        stream_handler.setFormatter(formatter)
        logger.addHandler(stream_handler)
    file_handler = logging.FileHandler('./log/SprintLog.log')
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

[...]

def sprint_commit(urls, verbose):
    log(verbose)
    csv_headers = ["Day", "Sprint_week", "Week", "Authors"]
    csv_headers_branches = ["Day", "Sprint_week", "Week", "Authors", "Commits"]
    repo_index = 0
    for url in urls:
        repo = Repository(path_to_repo=url).traverse_commits()
        commit = next(repo)
        logger.info(f'Project: {commit.project_name}')
        print(f'(sprint_week_all_commit) Project: {commit.project_name}')
        git = Git(commit.project_path)
        logger.debug(f'Project: {commit.project_name} #Commits: {git.total_commits()}')
        if verbose:
            log_view(url, commit.project_name, csv_headers)
        else:
            bar_view(url, commit.project_name, git.total_commits(), csv_headers)
        r = Repo(commit.project_path)
        remote_refs = r.remote().refs
        index = 1
```

```
40          for refs in remote_refs:
41              branch_name = refs.name.split('/')
42              if branch_name[len(branch_name) - 1] == 'HEAD':
43                  continue
44              print(f'(sprint_week_branch_commit) Project: {commit.project_name} Branch: {refs.name} '
45                    f'#: {index}/{len(remote_refs)-1}')
46              len_branch = len(list(Repository(path_to_repo=url, only_in_branch=refs.name).traverse_commits
        ()))
47              branch_view(url, refs.name, commit.project_name, len_branch, csv_headers_branches)
48              index += 1
49          repo_index += 1
50
51  [...]
52
53  def log_view(repo, repo_name, csv_headers):
54      author = []
55      with open("./data-results/sprint_week_" + repo_name + ".csv", 'w') as f:
56          writer = csv.DictWriter(f, fieldnames=csv_headers)
57          writer.writeheader()
58          week_commit = 0
59          prec_commit = None
60          for commit in Repository(path_to_repo=repo).traverse_commits():
61
62              logger.info(f'Hash: {commit.hash}, '
63                          f'Week: {commit.committer_date.isocalendar()[1]}, '
64                          f'Time: {commit.committer_date}, '
65                          f'Author: {commit.author.email}')
66
67              if prec_commit == None:
68                  prec_commit = commit
69                  week_commit = week_commit + 1
70                  author.append(commit.author.email)
71                  continue
72
73              if prec_commit.committer_date.year == commit.committer_date.year and \
74                      prec_commit.committer_date.isocalendar()[1] == commit.committer_date.isocalendar()
        [1]:
75                  week_commit = week_commit + 1
76                  if commit.author.email not in author:
77                      author.append(commit.author.email)
78              else:
79                  writer.writerow({csv_headers[0]: prec_commit.committer_date,
80                                   csv_headers[1]: week_commit,
81                                   csv_headers[2]: prec_commit.committer_date.isocalendar()[1],
82                                   csv_headers[3]: len(author)})
83                  week_commit = 1
84                  author = []
85                  author.append(commit.author.email)
86              prec_commit = commit
87          writer.writerow({csv_headers[0]: prec_commit.committer_date,
88                           csv_headers[1]: week_commit,
89                           csv_headers[2]: prec_commit.committer_date.isocalendar()[1],
90                           csv_headers[3]: len(author)})
91      logger.info(f'Sprint Week: {repo_name}')
92
93  [...]
94
95  SLIDING_WINDOW = 4
96  SCRUM_SEQUENCE = 1
97
98  partial_path = input("Enter CSV Repositories: data-results/sprint_week_")
99  path = "data-results/sprint_week_" + partial_path
100
101 path_split = path.split('/')
102
```

```
103 data = pd.read_csv(path)
104 year_week_x = [x[:4] + "-" + str(y) for x, y in zip(data['Day'], data['Week'])]
105
106 valid_sprint = []
107 if len(year_week_x) >= SLIDING_WINDOW and len(
108         year_week_x) - SLIDING_WINDOW + 1 > 0:
109     for i in range(len(year_week_x) - SLIDING_WINDOW + 1):
110         if statistics.mean(data['Sprint_week'][i:i + SLIDING_WINDOW - 1]) > data['Sprint_week'][i +
        SLIDING_WINDOW - 1]:
111             valid_sprint.append(year_week_x[i:i + SLIDING_WINDOW])
112 else:
113     print("Unable to obtain sprints: SLIDING_WINDOW > data to be analyzed.")
114     exit(0)
115
116 not_consecutive_sprint = []
117 for entry in valid_sprint:
118     year = 0
119     day = 0
120     for sprint in entry:
121         if year == 0 and day == 0:
122             year = int(sprint[0:4])
123             day = int(sprint[4:].replace('-', ''))
124         else:
125             if (year == int(sprint[0:4]) and day + 1 == int(sprint[4:].replace('-', '')) or
126                     year + 1 == int(sprint[0:4]) and day == 52 and int(sprint[4:].replace('-', '')) == 1)
        :
127                 a = 1
128             else:
129                 not_consecutive_sprint.append(entry)
130                 break
131             day = int(sprint[4:].replace('-', ''))
132             year = int(sprint[0:4])
133 legit_sprint = [x for x in valid_sprint if x not in not_consecutive_sprint]
134
135 def sprint_sequence(lista, ind):
136     if ind - 1 < 0:
137         return []
138     list_return = []
139     sprint_overlap = lista[ind - 1]
140     list_return.append(sprint_overlap)
141     for sprint in lista:
142         if len(intersection(sprint_overlap, sprint)) == 0:
143             list_return.append(sprint)
144             sprint_overlap = sprint
145     return list_return
146
147 if len(legit_sprint) < SLIDING_WINDOW:
148     exit("Unable to obtain sprints: SLIDING_WINDOW > legitimate data")
149 good_sprint_sequence = sprint_sequence(legit_sprint, SCRUM_SEQUENCE)
150 if len(good_sprint_sequence) < 0:
151     exit("Not enough data for a Sprint window (len(good_sprint_sequence) < 0)")
152
153 sprint_develop = np.zeros(len(year_week_x), dtype=int)
154 sprint_test = np.zeros(len(year_week_x), dtype=int)
155 sprint_else = data['Sprint_week'].copy()
156
157 for id, year_week in enumerate(year_week_x):
158     for sprint in good_sprint_sequence:
159         if [True for s in sprint[0:SLIDING_WINDOW - 1] if s == year_week]:
160             sprint_develop[id] = data['Sprint_week'][id]
161             sprint_else[id] = 0
162             break
163         if year_week == sprint[SLIDING_WINDOW - 1]:
164             sprint_test[id] = data['Sprint_week'][id]
165             sprint_else[id] = 0
```

A.2 Bag-of-Words model application on Sprint commit message

```
import logging
from src import ProgressionBar
from pydriller import Repository
from pydriller import Git
from spacy.matcher import Matcher
from nltk.stem.porter import *
import re
from nltk import ngrams
from nltk.corpus import stopwords
[...]

def log_view(repo, repo_name, csv_headers):
    """ Sprint Weeks: log console """
    with open("./data-results/bow_sprint_week_" + repo_name + ".csv", 'w') as f:
        writer = csv.DictWriter(f, fieldnames=csv_headers)
        writer.writeheader()
        msg_commit = ""
        prec_commit = None
        for commit in Repository(path_to_repo=repo).traverse_commits():

            logger.info(f'Hash: {commit.hash}, '
                        f'Week: {commit.committer_date.isocalendar()[1]}, '
                        f'Time: {commit.committer_date}, '
                        f'Messaggio: {commit.msg}')

            if prec_commit == None:
                prec_commit = commit
                msg_commit = commit.msg    # add msg commit
                continue

            if prec_commit.committer_date.year == commit.committer_date.year and \
                    prec_commit.committer_date.isocalendar()[1] == commit.committer_date.isocalendar()
      [1]:
                msg_commit = msg_commit +" "+ commit.msg
            else:
                writer.writerow({csv_headers[0]: prec_commit.committer_date,  # Day
                                 csv_headers[1]: prec_commit.committer_date.isocalendar()[1],  # Week
                                 csv_headers[2]: msg_commit})  # Msg_data
                msg_commit = commit.msg  #reset
            prec_commit = commit
        writer.writerow({csv_headers[0]: prec_commit.committer_date,  # Day
                         csv_headers[1]: prec_commit.committer_date.isocalendar()[1],  # Week
                         csv_headers[2]: msg_commit})  # Msg_data
    logger.info(f'BoW Sprint Week: {repo_name}')

REPLACE_BY_SPACE_RE = re.compile('[/(){}\[\]\|@,;]')
BAD_SYMBOLS_RE = re.compile('[^0-9a-z #+_]')
STOPWORDS = set(stopwords.words('english'))

partial_path = input("Enter CSV Repositories: data-results/bow_sprint_week_")
path = "data-results/bow_sprint_week_" + partial_path
path_split = path.split('/')

data_sprint = pd.read_csv(path)
data_sprint = data_sprint[pd.notnull(data_sprint['Msg_data'])]  # checking not missing msg

def clean_text(text):
    text = BeautifulSoup(text, "lxml").text
    text = text.lower()
    text = REPLACE_BY_SPACE_RE.sub(' ', text)
```

```
60      text = BAD_SYMBOLS_RE.sub(' ', text)
61      text = ' '.join(word for word in text.split() if word not in STOPWORDS)
62      return text
63
64  data_sprint['Msg_data'] = data_sprint['Msg_data'].apply(clean_text)
65
66  # STEMMING
67  stemmer = PorterStemmer()
68  csv_headers = ["Day", "Week", "Msg_data"]
69
70  [...]
71
72  # Most common word
73  data_sprint = pd.read_csv("stemmingbowset.csv")
74  msg_occurrences = []
75
76  for msg in data_sprint['Msg_data']:
77      for word in msg.split():
78          msg_occurrences.append(word)
79  occurrences = Counter(msg_occurrences)
80
81  # Top 10 BoW Frequency:
82  text_box = '#Top BoW Frequency'
83  conteggio = 0
84  for most_word in occurrences.most_common(20):
85      if not most_word[0].isdigit() and conteggio < 11:
86          text_box += '\n' + most_word[0] + ': ' + str(most_word[1])
87          conteggio += 1
88
89  # Top n-grams token:
90  tokenstr = ''
91  for token in msg_occurrences:
92      if not token.isdigit():
93          tokenstr += token + ' '
94  most_coulpe_token = Counter(list(ngrams(tokenstr.split(), 2)))
95
96  # Top 10 n-grams token:
97  text_box_pair = '#Top BoW Pair Token'
98  conteggio = 0
99  for most_word in most_coulpe_token.most_common(10):
100     text_box_pair += '\n' + str(most_word[0]) + ': ' + str(most_word[1])
101     conteggio += 1
102
103 data_sprint = pd.read_csv("stemmingbowset.csv")
104 nlp = spacy.load('en_core_web_sm')
105
106 m_tool = Matcher(nlp.vocab)
107 fix = [[{"LOWER": "fix"}],
108        [{"TEXT": {"REGEX": "^fix"}}]]
109 test = [[{"LOWER": "test"}],
110         [{"TEXT": {"REGEX": "^test"}}]]
111 bug = [[{"LOWER": "bug"}],
112        [{"TEXT": {"REGEX": "^bug"}}]]
113 debug = [[{"LOWER": "debug"}],
114        [{"TEXT": {"REGEX": "^debug"}}]]
115 refactoring = [[{"LOWER": "refact"}],
116                [{"TEXT": {"REGEX": "^refact"}}]]
117 feature = [[{"LOWER": "feature"}],
118            [{"TEXT": {"REGEX": "^feature"}}]]
119 documentation = [[{"LOWER": "documentation"}],
120                  [{"TEXT": {"REGEX": "^documentation"}}]]
121
122 m_tool.add('FIX', fix, on_match=None)
123 m_tool.add('TEST', test, on_match=None)
124 m_tool.add('BUG', bug, on_match=None)
```

```
125 m_tool.add('DEBUG', debug, on_match=None)
126 m_tool.add('REF', refactoring, on_match=None)
127 m_tool.add('FEAT', feature, on_match=None)
128 m_tool.add('DOC', documentation, on_match=None)
129
130 fieldnam = ['Day', 'Week', 'Tag']
131 [...]
132
133 for index, row in data_sprint.iterrows():
134     sentence = nlp(row['Msg_data'])
135     phrase_matches = m_tool(sentence)
136     for match_id, start, end in phrase_matches:
137         string_id = nlp.vocab.strings[match_id]
138         span = sentence[start:end]  # The matched span
139         if span.text:
140             with open("finale.csv", 'a') as f:
141                 writer = csv.DictWriter(f, fieldnames=fieldnam)
142                 writer.writerow({'Day': row['Day'], 'Week': row['Week'], 'Tag': string_id})
143             f.close()
144
145 # -----------------------
146 # bow + tag count
147 data_count = pd.read_csv('finale.csv')
148 data_count_head = ["Day", "Week", "Tag", "#Tag"]
149 week_grouped = data_count.groupby(["Day", "Week"])["Tag"].value_counts()
150 with open("bow_tag.csv", 'w') as f:
151     writer = csv.DictWriter(f, fieldnames=data_count_head)
152     writer.writeheader()
153 f.close()
154
155 bow = pd.DataFrame(week_grouped)
156 bow.to_csv("bow_tag.csv", header=False, mode="a")
157
158 # -----------------------
159 # bow tag filter multiple tag sprint to a single tag
160 data_count_filter = pd.read_csv('bow_tag.csv')
161
162 with open("bow_tag_filter.csv", 'w') as f:
163     writer = csv.DictWriter(f, fieldnames=data_count_head)
164     writer.writeheader()
165     for i, line in enumerate(data_count_filter['Day']):
166         if i == 0:
167             prec = line
168             max_index = i
169             continue
170         if prec == line:
171             if data_count_filter['#Tag'][i - 1] < data_count_filter['#Tag'][i]:
172                 prec = line
173                 max_index = i
174         else:
175             writer.writerow({'Day': data_count_filter['Day'][max_index], 'Week': data_count_filter['Week'
        ][max_index],
176                              'Tag': data_count_filter['Tag'][max_index], '#Tag': data_count_filter['#Tag'
        ][max_index]})
177             prec = line
178             max_index = i
179         if i == len(data_count_filter['Day']) - 1:
180             writer.writerow({'Day': data_count_filter['Day'][max_index], 'Week': data_count_filter['Week'
        ][max_index],
181                              'Tag': data_count_filter['Tag'][max_index], '#Tag': data_count_filter['#Tag'
        ][max_index]})
182 f.close()
```

Bibliography

[AAA17] Jasem M. Alostad, Laila R. A. Abdullah, and Lamya Sulaiman Aali. "A Fuzzy based Model for Effort Estimation in Scrum Projects". In: *International Journal of Advanced Computer Science and Applications* 8.9 (2017). DOI: 10.14569/IJACSA.2017.080939. URL: http://dx.doi.org/10.14569/IJACSA.2017.080939.

[Abr+17] Pekka Abrahamsson et al. "Agile software development methods: Review and analysis". In: *arXiv preprint arXiv:1709.08439* (2017).

[Aca+16] Yasemin Acar et al. "You get where you're looking for: The impact of information sources on code security". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 289–305.

[AG18] Abdullah Altaleb and Andrew Gravell. "Effort Estimation across Mobile App Platforms using Agile Processes: A Systematic Literature Review". In: July 2018. DOI: 10.17706/jsw.13.4.242-259.

[AGH17] Abdullah Aldahmash, Andy M Gravell, and Yvonne Howard. "A review on the critical success factors of agile software development". In: *European conference on software process improvement*. Springer. 2017, pp. 504–512.

[AH18] Alhejab Alhazmi and Shihong Huang. "A Decision Support System for Sprint Planning in Scrum Practice". In: *SoutheastCon 2018*. 2018, pp. 1–9. DOI: 10.1109/SECON.2018.8479063.

[Aha+18] Md Ahasanuzzaman et al. "Classifying stack overflow posts on API issues". In: *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2018, pp. 244–254.

[AHM06] John Anvik, Lyndon Hiew, and Gail C Murphy. "Who should fix this bug?" In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 361–370.

[Ama+16] Sven Amann et al. "MUBench: A benchmark for API-misuse detectors". In: *Proceedings of the 13th international conference on mining software repositories*. 2016, pp. 464–467.

[Ama+18] Sven Amann et al. "A systematic evaluation of static api-misuse detectors". In: *IEEE Transactions on Software Engineering* 45.12 (2018), pp. 1170–1188.

[Amb12] Scott Ambler. *Agile database techniques: Effective strategies for the agile software developer*. John Wiley & Sons, 2012.

[ARG06] Juan Jose Amor, Gregorio Robles, and Jesus M. Gonzalez-Barahona. "Effort Estimation by Characterizing Developer Activity". In: *Proceedings of the 2006 International Workshop on Economics Driven Software Engineering Research*. EDSER '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 3–6. ISBN: 1595933964. DOI: 10.1145/1139113.1139116. URL: https://doi.org/10.1145/1139113.1139116.

[AX09] Mithun Acharya and Tao Xie. "Mining API error-handling specifications from source code". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2009, pp. 370–384.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321278658.

[Bav16] Gabriele Bavota. "Mining unstructured data in software repositories: Current and future trends". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. IEEE. 2016, pp. 1–12.

[BBW13] Xu Ben, Shen Beijun, and Yang Weicheng. "Mining developer contribution in open source software using visualization techniques".

In: *2013 Third International Conference on Intelligent System Design and Engineering Applications*. IEEE. 2013, pp. 934–937.

[Bha+13] Pamela Bhattacharya et al. "An empirical analysis of bug reports and bug fixing in open source android apps". In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 133–143.

[Boe+09] Barry W Boehm et al. *Software cost estimation with COCOMO II*. Prentice Hall Press, 2009.

[BP10] Saleem Basha and Dhavachelvan Ponnurangam. "Analysis of empirical software effort estimation models". In: *arXiv preprint arXiv:1004.1239* (2010).

[BR08] Armin Beer and Rudolf Ramler. "The role of experience in software testing practice". In: *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. IEEE. 2008, pp. 258–265.

[Bro14] Meta S Brown. "Transforming unstructured data into useful information". In: *Big Data, Mining, and Analytics*. Auerbach Publications, 2014, pp. 227–246.

[BTB07] Bilge Baskeles, Burak Turhan, and Ayse Bener. "Software effort estimation using machine learning methods". In: *2007 22nd international symposium on computer and information sciences*. IEEE. 2007, pp. 1–6.

[CA16] João Caldeira and Fernando Brito e Abreu. "Software development process mining: Discovery, conformance checking and enhancement". In: *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE. 2016, pp. 254–259.

[Cat+19] Gemma Catolino et al. "How the Experience of Development Teams Relates to Assertion Density of Test Classes". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 223–234.

[CDB98] Bradford Clark, Sunita Devnani-Chulani, and Barry Boehm. "Calibrating the COCOMO II post-architecture model". In: *Proceedings*

of the 20th international conference on Software engineering. IEEE. 1998, pp. 477–480.

[CH01] Alistair Cockburn and Jim Highsmith. "Agile software development, the people factor". In: *Computer* 34.11 (2001), pp. 131–133.

[CLC03] David Cohen, Mikael Lindvall, and Patricia Costa. "Agile software development: A dacs state-of-the-art report". In: *Fraunhofer Center for Experimental Software Engineering Maryland and The University of Maryland* (2003).

[Coe+18] Jailton Coelho et al. "Identifying unmaintained projects in github". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM. 2018, p. 15.

[Coh05] Mike Cohn. *Agile estimating and planning*. Pearson Education, 2005.

[CS12] Krishna Kumar Chaturvedi and VB Singh. "Determining bug severity using machine learning techniques". In: *2012 CSI sixth international conference on software engineering (CONSEG)*. IEEE. 2012, pp. 1–6.

[DDP18] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. "FEVER: An approach to analyze feature-oriented changes and artefact coevolution in highly configurable systems". In: *Empirical Software Engineering* 23.2 (2018), pp. 905–952.

[Dej+11] Karel Dejaeger et al. "Data mining techniques for software effort estimation: a comparative study". In: *IEEE transactions on software engineering* 38.2 (2011), pp. 375–397.

[Dem+02] Bert J Dempsey et al. "Who is an open source software developer?" In: *Communications of the ACM* 45.2 (2002), pp. 67–72.

[DLR10] Marco D'Ambros, Michele Lanza, and Romain Robbes. "An extensive comparison of bug prediction approaches". In: May 2010, pp. 31–41. DOI: 10.1109/MSR.2010.5463279.

[Era+19] Haggai Eran et al. "Design Patterns for Code Reuse in HLS Packet Processing Pipelines". In: *2019 IEEE 27th Annual International*

Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE. 2019, pp. 208–217.

[Fer+20a] Rudolf Ferenc et al. "A public unified bug dataset for java and its assessment regarding metrics and bug prediction". In: *Software Quality Journal* 28.4 (2020), pp. 1447–1506.

[Fer+20b] Marta Fernández-Diego et al. "An Update on Effort Estimation in Agile Software Development: A Systematic Literature Review". In: *IEEE Access* 8 (2020), pp. 166768–166800. DOI: 10.1109/ACCESS.2020.3021664.

[FF00] Joseph Feller and Brian Fitzgerald. "A framework analysis of the open source software development paradigm". In: *ICIS 2000 proceedings of the twenty first international conference on information systems*. Association for Information Systems (AIS). 2000, pp. 58–69.

[GJM06] Stein Grimstad, Magne Jørgensen, and Kjetil Moløkken-Østvold. "Software effort estimation terminology: The tower of Babel". In: *Information and Software Technology* 48.4 (2006), pp. 302–310.

[GKS08] 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*. 2008, pp. 129–132.

[Gla+18] Elena L Glassman et al. "Visualizing API usage examples at scale". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12.

[Gor+18] Peter Leo Gorski et al. "Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse". In: *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. 2018, pp. 265–281.

[GPD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "An exploratory study of the pull-based software development model". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 345–355.

[GRF17] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. "Some from here, some from there: Cross-project code reuse in github". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, pp. 291–301.

[GS17] Georgios Gousios and Diomidis Spinellis. "Mining software engineering data from GitHub". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 501–502.

[GS20] Hamayoon Ghafory and Faqeed Ahmad Sahnosh. "The review of software cost estimation model: SLIM". In: *J. Adv. Academic Res.* 2.4 (2020), pp. 511–515.

[HE22] Abeer Hamdy and Gloria Ezzat. "Deep mining of open source software bug repositories". In: *International Journal of Computers and Applications* 44.7 (2022), pp. 614–622.

[JJE14] René Just, Darioush Jalali, and Michael D Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 437–440.

[JLW12] Woosung Jung, Eunjoo Lee, and Chisu Wu. "A survey on mining software repositories". In: *IEICE TRANSACTIONS on Information and Systems* 95.5 (2012), pp. 1384–1406.

[KCS13] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. "Is it all lost? A study of inactive open source projects". In: *IFIP international conference on open source systems*. Springer. 2013, pp. 61–79.

[Ker+22] Harold Kerzner et al. "Project Management: A Systems Approach to Planning, Scheduling, and Controlling". In: (2022).

[KFW18] Reni Kurnia, Ridi Ferdiana, and Sunu Wibirama. "Software Metrics Classification for Agile Scrum Process: A Literature Review". In: *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. 2018, pp. 174–179. DOI: 10.1109/ISRITI.2018.8864244.

[Kit+02] Barbara Kitchenham et al. "An empirical study of maintenance and development estimation accuracy". In: *Journal of systems and software* 64.1 (2002), pp. 57–77.

[KKM13] Jacky Keung, Ekrem Kocaguneli, and Tim Menzies. "Finding conclusion stability for selecting the best effort predictor in software effort estimation". In: *Automated Software Engineering* 20.4 (2013), pp. 543–567.

[KM09] Barbara Kitchenham and Emilia Mendes. "Why comparative effort prediction studies may be invalid". In: *Proceedings of the 5th international Conference on Predictor Models in Software Engineering*. 2009, pp. 1–5.

[LGS21] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. "A systematic review of API evolution literature". In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–36.

[LM09] Chris Lokan and Emilia Mendes. "Applying moving windows to software effort estimation". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2009, pp. 111–122.

[McI+16] Shane McIntosh et al. "An empirical study of the impact of modern code review practices on software quality". In: *Empirical Software Engineering* 21.5 (2016), pp. 2146–2189.

[MCM16] Martin Monperrus, Eduardo Campos, and Marcelo Maia. "Searching stack overflow for API-usage-related bug fixes using snippet-based queries". In: *26th Annual International Conference on Computer Science and Software Engineering*. 2016.

[Men+17] Tim Menzies et al. "Negative results for software effort estimation". In: *Empirical Software Engineering* 22.5 (2017), pp. 2658–2683.

[MJ03] Kjetil Moløkken-Østvold and Magne Jørgensen. "A review of surveys on software effort estimation". In: (2003).

[MJ15] Lech Madeyski and Marian Jureczko. "Which process metrics can significantly improve defect prediction models? An empirical study". In: *Software Quality Journal* 23.3 (2015), pp. 393–422.

[MKK09] Subhas Chandra Misra, Vinod Kumar, and Uma Kumar. "Identifying some important success factors in adopting agile software development practices". In: *Journal of systems and software* 82.11 (2009), pp. 1869–1890.

[Moe+15] Julie Moeyersoms et al. "Comprehensible software fault and effort prediction: A data mining approach". In: *Journal of Systems and Software* 100 (2015), pp. 80–90.

[MPR19] Reed Milewicz, Gustavo Pinto, and Paige Rodeghero. "Characterizing the roles of contributors in open-source scientific software projects". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 421–432.

[Mur+18] Emerson Murphy-Hill et al. "Discovering API usability problems at scale". In: *Proceedings of the 2nd International Workshop on API Usage and Evolution*. 2018, pp. 14–17.

[MY12] Leandro L Minku and Xin Yao. "Can cross-company data improve performance in software effort estimation?" In: *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*. 2012, pp. 69–78.

[Nie+21] Sebastian Nielebock et al. "Guided pattern mining for API misuse detection by change-based code analysis". In: *Automated Software Engineering* 28.2 (2021), pp. 1–48.

[PC14] Rashmi Popli and Naresh Chauhan. "Cost and effort estimation in agile software development". In: *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*. 2014, pp. 57–61. DOI: 10.1109/ICROIT.2014.6798284.

[PFM13] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. "An empirical study of API usability". In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 5–14.

[Pha+14] Raphael Pham et al. "Enablers, inhibitors, and perceptions of testing in novice software teams". In: *Proceedings of the 22nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering. 2014, pp. 30–40.

[PMR17] S. Muthu Perumal Pillai, S.D. Madhukumar, and T. Radharamanan. "Consolidating evidence based studies in software cost/-effort estimation — A tertiary study". In: *TENCON 2017 - 2017 IEEE Region 10 Conference* (2017), pp. 833–838.

[PSV11] Wouter Poncin, Alexander Serebrenik, and Mark Van Den Brand. "Process mining software repositories". In: *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE. 2011, pp. 5–14.

[Rad+13] Danijel Radjenović et al. "Software fault prediction metrics: A systematic literature review". In: *Information and software technology* 55.8 (2013), pp. 1397–1418.

[RD13] Foyzur Rahman and Premkumar Devanbu. "How, and why, process metrics are better". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 432–441.

[Red11] Martin Reddy. *API Design for C++*. Elsevier, 2011.

[RI19] Derek Reimanis and Clemente Izurieta. "Behavioral Evolution of Design Patterns: Understanding Software Reuse Through the Evolution of Pattern Behavior". In: *International Conference on Software and Systems Reuse*. Springer. 2019, pp. 77–93.

[RM12] Akif Raza and Hammad Majeed. "Issues and Challenges In Scrum Implementation". In: *International Journal of Scientific and Engineering Research* 3 (Aug. 2012).

[Rob+14] Gregorio Robles et al. "Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 222–231.

[Rob09] Martin P Robillard. "What makes APIs hard to learn? Answers from developers". In: *IEEE software* 26.6 (2009), pp. 27–34.

[Rol+18] Thomas Rolfsnes et al. "Aggregating association rules to improve change recommendation". In: *Empirical Software Engineering* 23.2 (2018), pp. 987–1035.

[RW01] Gene Rowe and George Wright. "Expert opinions in forecasting: the role of the Delphi technique". In: *Principles of forecasting*. Springer, 2001, pp. 125–144.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. "PyDriller: Python framework for mining software repositories". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. ISBN: 9781450355735. DOI: 10.1145/3236024.3264598. URL: http://dl.acm.org/citation.cfm?doid=3236024.3264598.

[SB02] Ken Schwaber and Mike Beedle. *Agile software development with scrum. Series in agile software development*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[SBS17] Apoorva Srivastava, Sukriti Bhardwaj, and Shipra Saraswat. "SCRUM model for agile methodology". In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE. 2017, pp. 864–869.

[Sha+09] Weiyi Shang et al. "Mapreduce as a general framework to support research in mining software repositories (MSR)". In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE. 2009, pp. 21–30.

[SMS16] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. "From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects". In: *Empirical Software Engineering* 21.2 (2016), pp. 642–683.

[SPH16] Federica Sarro, Alessio Petrozziello, and Mark Harman. "Multi-objective software effort estimation". In: *2016 IEEE/ACM 38th*

International Conference on Software Engineering (ICSE). IEEE. 2016, pp. 619–630.

[Spi17] Diomidis Spinellis. "A repository of Unix history and evolution". In: *Empirical Software Engineering* 22.3 (2017), pp. 1372–1404.

[SSK96] Martin Shepperd, Chris Schofield, and Barbara Kitchenham. "Effort estimation using analogy". In: *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE. 1996, pp. 170–178.

[Sty+08] Jeffrey Stylos et al. "A case study of API redesign for improved usability". In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE. 2008, pp. 189–192.

[Sty+09] Jeffrey Stylos et al. "Improving API documentation using API usage information". In: *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2009, pp. 119–126.

[Sun+16] Xiaobing Sun et al. "Mining software repositories for automatic interface recommendation". In: *Scientific Programming* 2016 (2016).

[Sve+19] Amann Sven et al. "Investigating next steps in static API-misuse detection". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 265–275.

[Tan+14] Lin Tan et al. "Bug characteristics in open source software". In: *Empirical software engineering* 19.6 (2014), pp. 1665–1705.

[Tei17] Jose Teixeira. "Release early, release often and release on time. an empirical case study of release management". In: *IFIP International Conference on Open Source Systems*. Springer, Cham. 2017, pp. 167–181.

[Thu+12] Ferdian Thung et al. "An empirical study of bugs in machine learning systems". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE. 2012, pp. 271–280.

[TJ14] Adam Trendowicz and Ross Jeffery. "Software project effort estimation". In: *Foundations and Best Practice Guidelines for Success, Constructive Cost Model–COCOMO pags* 12 (2014), pp. 277–293.

[TR16] Christoph Treude and Martin P Robillard. "Augmenting API documentation with insights from stack overflow". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 392–403.

[TS05] Pang-Ning Tan and Michael Steinbach. *e Kumar, V.(2005) Introduction to Data Mining*. 2005.

[UKR21] Gias Uddin, Foutse Khomh, and Chanchal K Roy. "Automatic api usage scenario documentation from technical q&a sites". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30.3 (2021), pp. 1–45.

[Unt+11] Michael Unterkalmsteiner et al. "Evaluation and measurement of software process improvement—a systematic literature review". In: *IEEE Transactions on Software Engineering* 38.2 (2011), pp. 398–424.

[UR15] Gias Uddin and Martin P Robillard. "How API documentation fails". In: *Ieee software* 32.4 (2015), pp. 68–75.

[Usm+14] Muhammad Usman et al. "Effort estimation in agile software development: a systematic literature review". In: *Proceedings of the 10th international conference on predictive models in software engineering*. 2014, pp. 82–91.

[Uus+15] Laura Uusitalo et al. "An overview of methods to evaluate uncertainty of deterministic models in decision support". In: *Environmental Modelling & Software* 63 (2015), pp. 24–31.

[Wen+19] Ming Wen et al. "Exposing library API misuses via mutation analysis". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 866–877.

[WH05a] C. C. Williams and J. K. Hollingsworth. "Automatic mining of source code repositories to improve bug finding techniques". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 466–480. DOI: 10.1109/TSE.2005.63.

[WH05b] Chadd C Williams and Jeffrey K Hollingsworth. "Automatic mining of source code repositories to improve bug finding techniques".

In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 466–480.

[WJ99] Fiona Walkerden and Ross Jeffery. "An empirical study of analogy-based software effort estimation". In: *Empirical software engineering* 4.2 (1999), pp. 135–158.

[WMJ10] Ye-Chi Wu, Lee Wei Mar, and Hewijin Christine Jiau. "CoDocent: Support API usage with code example and API documentation". In: *2010 Fifth International Conference on Software Engineering Advances*. IEEE. 2010, pp. 135–140.

[WOM15] Peter A Whigham, Caitlin A Owen, and Stephen G Macdonell. "A baseline model for software effort estimation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.3 (2015), pp. 1–11.

[Won+16] W Eric Wong et al. "A survey on software fault localization". In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[Zha+19] Jingxuan Zhang et al. "Enriching API documentation with code samples and usage scenarios from crowd knowledge". In: *IEEE Transactions on Software Engineering* 47.6 (2019), pp. 1299–1314.

[Zim06] Thomas Zimmermann. "Taking Lessons from History". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 1001–1005. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134474. URL: http://doi.acm.org/10.1145/1134285.1134474.

[Živ+11] Jovan Živadinović et al. "Methods of effort estimation in software engineering". In: *Proc. Int. Symposium Engineering Management and Competitiveness (EMC)*. 2011, pp. 417–422.