



**Trinity College Dublin**

Coláiste na Tríonóide, Baile Átha Cliath

The University of Dublin

School of Computer Science and Statistics

**Sociometrics in Software Engineering:  
Measuring Technical Skills of Software  
Engineer through GitHub**

Rohan Bagwe

Supervisor: Prof. Stephen Barrett

September 7, 2020

A Final Year Project submitted in partial fulfilment  
of the requirements for the degree of  
Master of Science in Computer Science - Intelligent Systems

# Declaration

I, Rohan Dilip Bagwe, declare that the work described in this dissertation is entirely my work and to the best of my knowledge it contains no material previously published or submitted as an exercise for a degree at this or any other university except where due acknowledgement is made in the thesis.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at <http://www.tcd.ie/calendar>.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at <http://tcd-ie.libguides.com/plagiarism/ready-steady-write>.

Signed: Rohan Bagwe

Date: September 7, 2020

# Permission to Lend and/or Copy

I, *Rohan Dilip Bagwe*, agree that Trinity College Library may lend or copy this thesis upon request.

Signed: Rohan Bagwe

Date: September 7, 2020

# Acknowledgements

It gives me huge gratification in expressing my sincere thanks and appreciation to my esteemed guide ‘Prof. Stephen Barrett’ for his constant motivation, enthusiasm, patience, and immense knowledge. His guidance and unwavering support helped me resolve all challenges faced during the research.

To all the professors at ‘Trinity College Dublin’ who taught me so well, I express my ardent gratitude. You all have been a constant source of inspiration and admiration.

Also, special thanks to ‘Trinity College Dublin’ for the support shown during unfortunate situations of the COVID-19 pandemic.

I wholeheartedly thank my beloved parents and all family members for their unconditional love, support, care, and prayers.

Last but not the least, heartfelt thanks to all my friends in Dublin and India who helped and encouraged me throughout this journey.

# Abstract

Any software project relies gravely on the skill set of its engineers. Technical skill measurement and analysis are the key factors to observe the growth and performance of an engineer in a specific project. While the prevailing system does understand the utmost importance of skill measurement, the qualitative nature of it poses a challenge that has not met an optimized solution yet. So, how are the companies dealing with it, noticeably analyzing the self-authored resumes or LinkedIn profiles?

Many studies in the field to record the severity of the challenge have revealed that the primary issue faced by most recruiters when hiring engineers was to find sufficiently qualified candidates. Indeed, the majority of miss-hires in tech recruiting come down to an inaccurate measurement of the technical skills of the candidate. And even after the company has hired the engineers, they do not have quantified measurable parameters to assess their skill level. There is a need for more reliable metrics to evaluate engineers' technical skills and certify their knowledge on a particular subject. This could help management to assign the best resources to the project as per the budget and technical requirements of the task.

Traditionally, for the hiring process, the metrics used to evaluate the technical skills of a software engineer included education level, professional experience, online coding challenges, onsite interviews, and after the actual hiring, metrics change to the number of bugs fixed, number of new features developed or even the number of hours worked. These proxy variables do not offer any real or actionable insights to quantify the qualitative skills of individuals.

This research seeks to build a data model by capturing the qualitative signals from GitHub. This data model represents an opinionated view for measuring the technical skills of a software engineer. Using this conceptualized data model, this research also seeks to build an expert system that could rank order software engineers using a relative grading system. To evaluate the efficiency of the proposed solution, a sanity test is performed using manual evaluation by the human expert (the author of this research). The results indicate that the data model provides a reasonably accurate technical skill assessment of software engineers in Java technology by rank-ordering the contributors of the OSS Spring Boot project. We do see some loss in the conceptual fidelity of the proposed system. Thus, the limitations have also been discussed in the latter part of the thesis.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Research Problem . . . . .	2
1.3	Research Challenge . . . . .	2
1.4	Research Question . . . . .	3
1.5	Research Objectives . . . . .	3
1.6	Overview of Research Approach . . . . .	4
1.7	Dissertation Structure . . . . .	5
<b>2</b>	<b>State of the Art</b>	<b>6</b>
2.1	Background . . . . .	6
2.2	Developer Capability Assessment Model . . . . .	8
2.3	The Rasch Measurement Model . . . . .	10
2.4	Software Metrics . . . . .	10
2.4.1	Halstead’s Software Science . . . . .	11
2.4.2	McCabe Cyclomatic Complexity . . . . .	12
2.5	Online Skill Evaluation Platforms . . . . .	12
2.5.1	GitHub’s Interface . . . . .	12
2.5.2	Hacker Rank . . . . .	13
2.5.3	Waydev . . . . .	13
2.6	Code Analysis Tools . . . . .	14
2.6.1	SonarQube . . . . .	14
2.6.2	Lint . . . . .	14
2.7	Summary . . . . .	15
<b>3</b>	<b>Design and Methods</b>	<b>16</b>
3.1	Creation of Data Model using GitHub Signals . . . . .	16
3.1.1	GitHub API . . . . .	17
3.1.2	Data Preparation . . . . .	18
3.2	Measuring Technical Skills of a Software Engineer . . . . .	19

3.2.1	Using Code Commit Signal . . . . .	19
3.2.1.1	Overview . . . . .	19
3.2.1.2	Implementation . . . . .	20
3.2.1.3	Result . . . . .	20
3.2.1.4	Discussion . . . . .	20
3.2.2	Using Code Change Signal . . . . .	21
3.2.2.1	Overview . . . . .	21
3.2.2.2	Implementation . . . . .	22
3.2.2.3	Result . . . . .	23
3.2.2.4	Discussion . . . . .	24
3.2.3	Using Library Usage Signal . . . . .	25
3.2.3.1	Overview . . . . .	25
3.2.3.2	Implementation . . . . .	26
3.2.3.3	Result . . . . .	27
3.2.3.4	Discussion . . . . .	27
3.2.4	Using Code Churn Signal . . . . .	30
3.2.4.1	Overview . . . . .	30
3.2.4.2	Implementation . . . . .	31
3.2.4.3	Result . . . . .	32
3.2.4.4	Discussion . . . . .	33
3.2.5	Using Code Stability Signal . . . . .	34
3.2.5.1	Overview . . . . .	34
3.2.5.2	Implementation . . . . .	35
3.2.5.3	Result . . . . .	35
3.2.5.4	Discussion . . . . .	36
3.3	Measuring Technical Skills using Data Model . . . . .	37
3.4	Measuring Technical Skills using Manual Evaluation . . . . .	38
3.5	Ranking Mechanism of Expert System . . . . .	40
3.5.1	Overview . . . . .	40
3.5.2	Measuring Technical Skills . . . . .	41
3.6	Expert System Design . . . . .	41
3.6.1	Creation of Technical Skills Assessment Dashboard . . . . .	41
<b>4</b>	<b>Result and Evaluation</b>	<b>45</b>
4.1	Data Collection . . . . .	45
4.2	Software Engineers Ranking by Expert System . . . . .	45
4.3	Software Engineers Ranking by Manual Evaluation . . . . .	47
4.4	Comparing Technical Skills of Software Engineers . . . . .	48

<b>5</b>	<b>Conclusion</b>	<b>50</b>
5.1	Discussion . . . . .	50
5.2	Research Contribution . . . . .	51
5.3	Future Work . . . . .	52
<b>A1</b>	<b>Code Commit and Code Change Signal</b>	<b>60</b>
A1.1	What is Commit? . . . . .	60
A1.2	What is Code Change? . . . . .	60
<b>A2</b>	<b>Measuring Library Usage Signal</b>	<b>62</b>
<b>A3</b>	<b>Measuring Code Churn Signal</b>	<b>63</b>
<b>A4</b>	<b>Measuring Code Stability Signal</b>	<b>64</b>
<b>A5</b>	<b>Measuring Technical Skills of Developer ‘A’ using Manual Evaluation Framework</b>	<b>65</b>



# List of Figures

1.1	Research Approach. . . . .	4
2.1	Developer capability assessment model [1]. . . . .	9
3.1	Variation in the total number of commits performed by ‘A’ per month . .	21
3.2	Total Number of LOC changed by ‘A’ . . . . .	23
3.3	Inclusion of blank lines in the code change signal extracted from GitHub	24
3.4	Steps involved in measuring Library Usage Signal . . . . .	27
3.5	Library usage of developer ‘A’ . . . . .	28
3.6	Detailed analysis of A’s library usage signal for AssertJ library . . . . .	28
3.7	Detailed analysis of A’s library usage signal for Util library . . . . .	28
3.8	Overall library usage by developer ‘A’ . . . . .	29
3.9	Variation in the total number of LOC added by ‘A’ vs. total number of LOC churned (or deleted) as of June 19, 2020 . . . . .	33
3.10	Variation in A’s code churn rate signal . . . . .	33
3.11	Variation in the LOC added by ‘A’ vs. LOC survived as of June 19, 2020	36
3.12	Code stability credits earned by ‘A’ every month . . . . .	36
3.13	Data model to measure technical skills of a software engineer . . . . .	38
3.14	Evaluation framework for manual analysis . . . . .	39
3.15	Architecture of the expert system . . . . .	42
3.16	Technical skill assessment dashboard 1 . . . . .	43
3.17	Technical skill assessment dashboard 2 . . . . .	43
3.18	Technical skill assessment dashboard 3 . . . . .	44
4.1	Comparing code commit signal of all software engineers (June 2018 - June 2020) . . . . .	48
4.2	Comparing code churn signal of all software engineers (June 2018 - June 2020) . . . . .	49
4.3	Comparing code stability signal of all software engineers (June 2018 - June 2020) . . . . .	49

A1.1 Code Commit Signal on GitHub UI [2] . . . . .	60
A2.1 Extracting Library Usage Signal [2] . . . . .	62
A3.1 Code Churn Example [2] . . . . .	63
A5.1 Different variety of commits performed by ‘A’ . . . . .	65
A5.2 Documentation performed by ‘A’ . . . . .	66
A5.3 Inefficiency of custom parsers to extract complex method calls: Method Chaining . . . . .	67
A5.4 Inefficiency of custom parsers to extract complex method calls: Method Referencing using ‘::’ operator . . . . .	67

# List of Tables

3.1	Research platform and scope of analysis. . . . .	19
3.2	Data model with quantified signals representing technical skills of developer ‘A’ . . . . .	40
4.1	Data model representing the quantified values of extracted signals for all software engineers . . . . .	45
4.2	Data model representing the highest value of every qualitative signal . . . . .	46
4.3	Technical skill measurement by expert system . . . . .	47
4.4	Rank ordering of software engineers by expert system . . . . .	47
4.5	Technical Skill assessment points by expert system and manual evaluation . . . . .	48
A5.1	Technical skill points of developer ‘A’ measured through manual evaluations . . . . .	67

# Abbreviations

<b>API</b>	Application Programming Interface
<b>CSS</b>	Cascading Style Sheets
<b>COCOMO</b>	Constructive Cost Model
<b>CSV</b>	Comma-separated values
<b>ETL</b>	Extract, Transform, Load
<b>FOSS</b>	Free and Open Source Software
<b>ICT</b>	Information and Communication Technology
<b>IP</b>	Internet Protocol
<b>ISO</b>	International Organization for Standardization
<b>JDK</b>	Java Development Kit
<b>LOC</b>	Lines of Code
<b>MISRA</b>	Motor Industry Software Reliability Association
<b>OECD</b>	Organisation for Economic Co-operation and Development
<b>OSS</b>	Open-source Software
<b>POC</b>	Proof of Concept
<b>REST</b>	Representational state transfer
<b>RQ</b>	Research Question
<b>SCM</b>	Source Code Management
<b>SHA</b>	Secure Hash Algorithms
<b>SME</b>	Subject Matter Expert
<b>SLOC</b>	Source Lines of Code
<b>SDLC</b>	Software Development Life Cycle
<b>TV</b>	Television
<b>URL</b>	Uniform Resource Locator
<b>UI</b>	User Interface
<b>VCS</b>	Version Control System
<b>VS</b>	Versus

# 1 Introduction

This chapter illuminates the motivation behind the research and explains the research question, challenges, and objectives. It includes the explanation and discussion of the technical approach proposed for this particular research question and also consists of a brief outline of the overall structure of the thesis.

## 1.1 Motivation

The revolutionization of computer science has enabled it to be a part of every aspect of our lives. Beginning from a simple calculator to life-preserving systems in hospitals, a computer software has become a crucial part of our technically advanced lives. Almost every sector in our civilization, including but not limited to health, defense, banking, aviation, and space exploration is now dependent on software. Thus, increasing dependency on software makes them crucially important to be of high quality. While quality is important to customers, so is the time. Customers tend to ask for high-quality solutions in a limited time. Such situations demand high-quality software with less rework and fewer bugs. These high-quality software solutions are majorly dependent on the technical skills of its developers. Thus, such a market raises a concern for the skill attestation of its creators to ensure the software quality.

Ferguson [3] defines ability as *“a trait defined by what an individual can do.”* According to Pear [4], skill is a specialized type of human abilities that is well organized, goal-oriented, and improves with practice. Campbell et al. [5] consider **‘technical skill’** as one of the eight major components of job performance. Hawk et al. [6] ranked technical skill as the most important skill for a software engineer. Also, after analyzing 785,325 projects or tasks at a global outsourcing marketplace Jørgensen [7] concluded that the risk of project failure may be reduced considerably if clients focus more on provider skills than low prices.

Traditionally, we have developed certain procedures and practices to follow that can help us reach the desired level of efficiency while measuring technical skills for on-boarding new engineers or assigning existing engineers to software projects. The current practices

in industry and research rely mostly on inspection of proxy variables such as candidate's education and experience on self-authored resumes, standardized online tests or coding challenges, self-authored portfolios on social networks such as LinkedIn [8]. Although these methods may indicate an individual's level of technical skill, they fail to quantify the qualitative nature of it. There is a need for more scientific software estimation methods to measure metrics such as the technical skills of a software engineer. Thus, the aforementioned limitations in technical skill assessment became a primary source of motivation for conducting this research.

## 1.2 Research Problem

Many researchers [9], [10] have faced problems to quantify the qualitative aspects of software development. A valid and scientific measure of technical skills of a software engineer that can be scored and interpreted would fill in the void of today's system. Thus, the overall research problem is to understand the extent to which we can measure the technical skills of a software engineer.

## 1.3 Research Challenge

The skill of an individual is something that can be visualized as the output of activity from the given input. Software industry relies heavily on the skill set of its developers. Many debates and discussions continue to go on about the efficiency of the software development processes. Most of these arguments have not been resolved to the date owing to the fact that our software industry lacks the basic element of quantifying the qualitative traits of the software engineers. To sum up, we do not have a reasonable metric to measure productivity among individual team members. Currently, the software industry lacks more reliable and comprehensive methods or tools to measure the productivity of a software engineer.

Factors including but not limited to the level of complexity of the task and the time required to complete the task make the technical skill assessment highly contextual. Even though it has been proved a tedious process researchers did not stop trying to find a way out. The famous quote by DeMarco [11] in business management says,

*"if you can't measure it, you can't manage it"*

but still, we do see unmeasurable traits of candidates being managed by businesses in almost every sector. Since measuring the skill is a contextual activity, It highly depends on the context of the business requirement. For example, why do we need to measure the candidate skill, is it for new hiring or allocating existing employees to the top account

projects across research and development teams within an organization? These situations make a bit tricky to measure productivity of an individual when we do not have a standard measuring metric. Though the number of features delivered per iteration gives us an estimate of the team effort when compared to other teams, it still does not help on the individual level. Individually, everyone plays a significant role in delivering the expected output by a team. When someone is responsible for implementing the features on the coding level, some might have designed an efficient way to implement that feature, some might have tested those features for any failure and someone like a Subject Matter Expert (SME) might have guided all those who involved in designing, implementing and testing the feature. Being a member on the team, one can visualize the individual effort but for any outsider, it is very hard to quantify the individual effort as it is completely subjective and depends on the context behind the skill measurement. There is a need to understand the available metrics and their context for measuring any kind of skill.

In this research, the focus was to develop a data model for measuring technical skills of a software engineer on certain assumptions. It has been observed that there cannot be one perfect solution to assess the technicality in all software development environments. Thus, the data model developed is based on an opinionated view of the expert (in this research, the author himself).

## 1.4 Research Question

Since our initial research scope was to investigate if it possible to measure technical skills of a software engineer, with the popularity of GitHub [12] in Open-source software (OSS) and its open dataset access to perform large scale exploratory analysis, the following research question has been formulated:

**RQ:**

*“Can we construct a data-model representing an opinionated view to measure technical skills of a software engineer through signals obtained from an Open-source software project on GitHub?”*

## 1.5 Research Objectives

We have defined the following research objectives to answer the research question discussed above:

1. To create a sophisticated data model for measuring technical skills of a software engineer as opposed to traditional approaches of skill assessment.

2. To discover useful qualitative signals from GitHub to construct a data model.
3. To develop an expert system that would rank order technical skills of software engineers based on an opinionated view of a data model.
4. To compare the efficiency of the proposed data model by comparing the technical skill points generated by the tool with the points estimated by manual evaluation.
5. To develop a dashboard visualization to compare technical skills of the software engineers within a team or a project.

## 1.6 Overview of Research Approach

The research approach for this thesis is explained as follows:

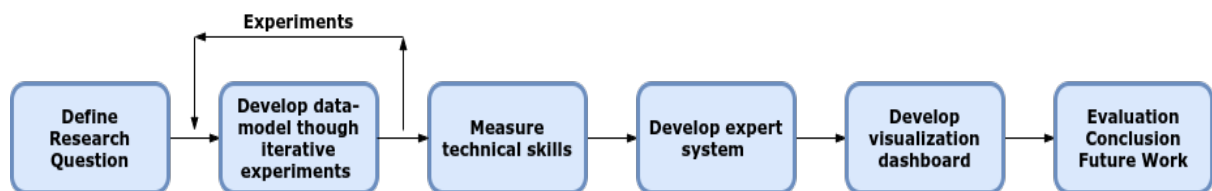


Figure 1.1: Research Approach.

- **Define the research question:** An extensive research in the domain of sociometric in software engineering was performed to understand the current state-of-the-art approaches of measuring technical skills of a software engineer. Based on the identified gaps in the state of the art, research questions for the thesis had been defined.
- **Develop data model through iterative experiments:** A series of experiments were performed to identify the qualitative signals from GitHub, to be used in the data model. Each successive experiment explored a deeper layer to refine the existing data model by extracting more meaningful signals.
- **Measure technical skills:** After developing a data model, an opinionated view of the extracted signals was defined to measure the technical skills of a software engineer.
- **Develop expert system:** Based on the opinionated view of the data model, an expert system was developed to rank order technical skills of software engineers.
- **Develop visualization dashboard:** A visualization dashboard was created to compare rankings of different software engineers. Also, visualizations of each qualitative signal incorporated in the data model were displayed on the dashboard.
- **Evaluation, Conclusion, Future Work:** The proposed expert system was evaluated to judge the correlation between technical skill points generated by the au-



tomated expert system and the points given by the human expert through manual analysis. The assessment of research objectives was performed and the limitations of the proposed system were also discussed.

## 1.7 Dissertation Structure

This section outlines the structure of the thesis.

**Chapter 1** introduces the reader with the need and motivation behind this research. It continues to elaborate on the research question along with the research objectives, various challenges faced and gives an overview of the research approach and dissertation structure.

**Chapter 2** discusses state of the art that starts with the background details about software development and challenges to quantify the technical skills of a software engineer. It then discusses similar studies which tried to measure the technical skills of software engineers with the metrics used in traditional systems.

**Chapter 3** discusses the design and methods used to achieve thesis objectives. Each experiment discusses its implementation, outcomes, limitations, and every successive experiment tries to overcome the shortcomings of the previous experiment. It also discusses the final design of the expert system and the associated shortcomings.

**Chapter 4** throws light on the collected data and discusses the approach used for the technical skill evaluation of software engineers.

**Chapter 5** concludes the research paper and discusses the research contribution, challenges, and future work.

## 2 State of the Art

With a wide pool available for the design and methodologies in today's market, many options had to be tried to reach an opinionated, feasible, and optimal solution to the question posed by this research. The options chosen to find an optimal answer to this thesis question were selected by venturing and exploring many domains namely socio-metric, expert systems, GitHub data mining, and relative grading. The main target of this chapter is to discuss various state-of-the-art approaches that were along on the same lines of this thesis, including a brief opening about the domains and methodologies used and the effect of their results in general.

### 2.1 Background

Software development is a structured process divided into multiple stages so that different teams can collaborate and finish the inclusive tasks associated with each stage. The responsibility of execution of individual tasks, to deliver quality software in time and cost, lies in the hands of developers, testers, designers, managers, etc. It is of utmost importance to select the team members after validating their skills and experience in the relevant field as per the project requirements. Poor technical skills of engineers may lead to faulty software or product delivery beyond time and cost allotted. Thus, the skill evaluation of the software engineer before assigning any project is crucial. This section specifies various measures, researches, and experiences provided by earlier researchers to evaluate the technical skills of a software engineer.

According to Tejaswini et al. [13], product quality and developer productivity are interconnected in software engineering. Thus, in the software industry, a developer who writes reusable, maintainable, efficient, and stable codes is of incredible value. But, due to the lack of technical skill assessment tools, organizations face many challenges in identifying and monitoring such software engineers. The authors [13] proposed a novel method and tool that not only measures the software engineer's productivity but also assist them to improve it. The productivity assessment tool considered parameters such as usage of

memory and machine cycles to provide a rating for a program written in C language. The proposed tool also identified bad programming constructs such as dead code, over-complicated expressions, complicated loops, improper scope definition, usage of pointers, and possible bugs. The authors suggested that such a tool could help organizations or educational institutes to identify their best engineers or students in terms of code efficiency and usage of industry standards.

Li et al. [14] discovered 53 essential attributes required by a software engineer. The research was performed at Microsoft by interviewing 59 engineers across 13 different divisions. These attributes were categorized by the engineer's personality, decision-making ability, and impact on team and product.

Lee et al. [15] predicted the expertise level of developers and task difficulty using data from psycho-physiological sensors such as electroencephalography and an eye-tracker. The authors tried to correlate psycho-physiological responses in software developers with matters of mental strain, effort, or expertise. The data collected from these sensors represented eye movements according to the velocity of shifts in the eye direction of 38 expert and novice software engineers. The authors built three Support Vector Machine models: the first two models using the data from an eye-tracker and electroencephalographic sensor respectively, and the third model using both the data sources combined. Data from these two sensors could predict the task difficulty and level of expertise with 64.9 and 97.7% precision and 68.6 and 96.4% recall, respectively.

Somasundaram et al. [16] developed a novel solution to classify the competency level in a Java programming language using Analytic Hierarchy Process. In this research, the authors assigned grades to validate students' programming skills using multi-criteria based assessment.

The work of Li et al. [17] focused on the influences of human factors on programming performance. With the help of statistical analysis, the authors analyzed the performance of the programmer. The authors discovered a correlation between a human personality type and programming performance. Programs developed by students with perceiving personality type (self-reported) executed a bit faster than the judging personality type.

Reinstedt, R. N [18] performed a test-based programmer performance prediction study on 534 programmers from 24 participating companies. A test battery composed of the IBM Programmer Aptitude Test and the Test of Sequential Instructions was administered to discover the relationship between job performance and the cognitive strength, vocational interest, and logical reasoning of the programmers.

Fitria et al, [19] developed a model to select team members based on the interdependency of skills using an artificial bee colony algorithm. This algorithm analyzed the strength

and risk of team resources to avoid the risk of failure in a software development project due to the lack of competent resources.

The authors [20] developed a model that scores the programming level of Java developers. In this research, authors defined the context information based tree structure to label the level of expertise of Java developers.

European e-Competence Framework (e-CF) [21] helps both the private and public businesses and organizations to form an informed decision while performing the technical evaluation of the ICT professionals. The e-CF has 41 competencies which are classified as per the five main ICT business areas; Plan, Build, Run, Enable, and Manage that relate to the European Qualifications Framework [22]. This framework assesses the technical skills of individuals with a specific focus on maintaining, developing, and managing ICT processes and projects. It helps to maintain common standards for skill and proficiency levels across entire Europe.

## 2.2 Developer Capability Assessment Model

According to [1], the quality of the software development process is affected by the technical knowledge, domain expertise, and experience of a programmer. Owing to the importance of the technical skill of the programmers, this research put forward a composite metric suite to describe the professional traits of the software engineer. This study divided the technical aspects of the programmers according to their knowledge, experience, learning ability, coding knowledge of the project platform, and the capabilities to identify the bugs and to develop the bug-free software system. Using these aspects of the programmer's knowledge, the author provided five metrics namely technical capability, experience, bug resistivity, coding capability, and learning interests of programmers to measure their technical skills. All of these metrics were defined by multiple comprehensive features to understand the associated aspect of each individual metric. For example feature such as the technical knowledge included knowledge of programming language, platform, and tools. The experiment performed to establish the composite metrics was conducted by performing a survey on five teams each having 30 programmers.

In figure 2.1, the metrics-suite measured the programmer's capability under different senses including metrics such as technical-capability, experience, coding capability, bug-resistive capability, and learning interest to measure the professional image of a programmer. Each metric was based on various sub-metrics. Technical Capability (Tc), measures the developer's ability to work efficiently on core and related technologies, tools, and platforms. Hands-on experience marks the highest rating for any developer. It can be markedly distinguished by the programmer or the team lead, which can help to assign a

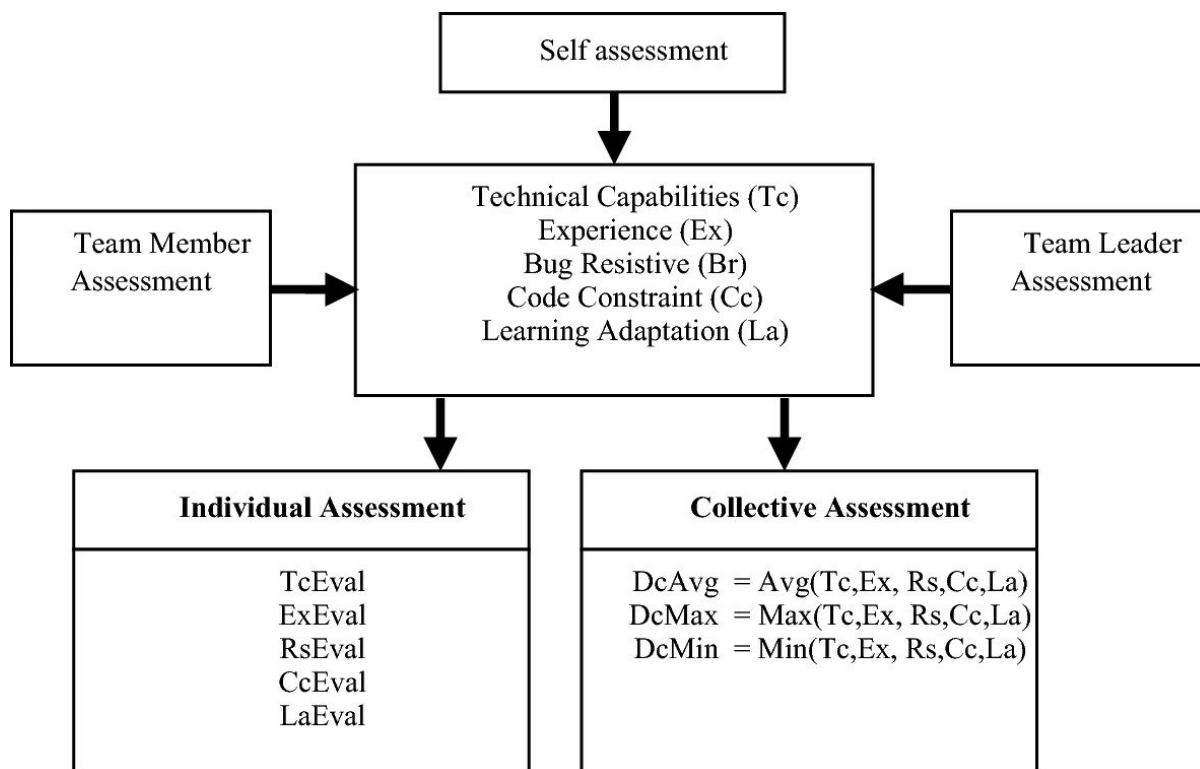


Figure 2.1: Developer capability assessment model [1].

suitable role and responsibilities. The Language hands-on (LHo), Platform Expertise (Pe), Tool Knowledge (Tk), Cross-Platform Programming (CPp), and Individual Application Design (IAd) were the sub-metrics discussed and evaluated under technical capability evaluation criteria. The aggregate operators were also implied on computed features to classify the programmers based on the capabilities. These metrics were evaluated analytically, with three different viewpoints, based on the rating given by the team members, programmer, and the team leader. The scoring was based on the inclusive sub-metrics as mentioned in the main metrics of the developer's capability. All team members, including the lead, were questioned individually and asked to rate an individual between 1 and 5 based upon the strength for that attribute. The rating of each co-programmer was measured against the precise distinctive weights, which was assigned as per the metric, its dependency and knowledge of team members. Then, the evaluated metric weights were applied under high-level capability metrics to measure the programmer's strength for that feature. Finally, the aggregate operators were applied to evaluate the capability of the programmer.

To conclude the model, a rule-based criteria was defined to differentiate the expert, skilled, and low-performance programmers. The technical capabilities of the programmers were rated from 0 and 1. All the programmers with technical capability over 0.7 were recognized as expert programmers while those with less than 0.4 were recognized as naive programmers. The author observed that no single programmer was perfect for satisfying

all technical requirements. Thus, some of the technical requirements of the project were observed to be fulfilled by group members in collective.

## 2.3 The Rasch Measurement Model

According to Bergersen et al. [23], skill is examined as a type of ability with specific characteristics. The authors focused on constructing an instrument to measure the technical skill of an individual based on a set of 19 Java programming tasks. In this study, both theoretical and practical perspectives enabled extensive empirical testing to identify professional programmers with the capability to develop high-quality software in a short time. To generalize the instrument, the scope of this thesis was limited to the tasks that do not belong to any particular domain or software technology.

Many existing models do assess the psychological abilities such as a skill by presenting questions to the individual and summing the scores of all the responses. In this research, the authors used the the Rasch measurement model to construct a valid instrument for assessing the technical skills of a programmer.

Rasch model in its original sense is a type of Item Response Theory (IRT) [24] model by which skill can be measured. This model resembles an additive conjoint measurement [25]. The Rasch measurement model uses maximum likelihood function to estimate the ability of the person and the difficulty level of the task. The model uses an interval-logit scale as the unit of measurement [23].

The Rasch model has been widely used in educational testing programs, such as OECD's Programme [26] for International Student Assessment. This model has also been used to measure developer's programming ability in C [27], Pascal [28], and Lisp [29] languages.

## 2.4 Software Metrics

Honglei et al. [30]. defines Software Metrics as

*“metrics that provide a measurement for the software and the process of software production. It is giving quantitative values to the attributes involved in the product or the process.”*

According to Norman Fenton [31] measurement is

*“a process by which numbers or symbols are assigned to attributes in the real world in such a way as to describe them according to clearly defined rules.”*

In the world of software systems, a direct relation can be established between software metrics and measurement. Software metrics are a quantitative guide to the performance

of certain aspects of the software, be it software processes, code characteristics such as cohesion, coupling, abstraction, or complexity parameters such as time and space[30]. The process of software development is highly volatile because requirements and methodologies change constantly over the Software Development Life Cycle (SDLC). Though this emphasizes the need for quantifiable metrics, it becomes difficult to measure the constantly changing software.

Software metrics allow programmers to refactor their code and determine its quality standard as per the requirements. To cite an example, a programmer might need to redesign the loop structure or control flow if the time complexity of the module exceeds the limit of 2 milliseconds. All measures are taken to accelerate the quality development of the software in the allocated time and budget. According to the research [32], software metrics can also be categorized into different groups as per the different measures performed on the code such as metrics of control flow and data flow complexity, object oriented metrics, safety and hybrid metrics.

Metrics play various roles in different stages of SDLC, for example at the beginning of the development cycle they might be used to determine the cost and working time of resources, during the design phase they might be used to measure the size of the software using Lines of Code (LOC) metric and after the development, complexity metrics could be used to estimate the code quality. Two considerable contributions in this area are Halstead's complexity [33] and McCabe cyclomatic complexity [34]. Both metrics utilize the premise that software complexity is strongly related to the measurable properties of the software.

### **2.4.1 Halstead's Software Science**

Software Metrics provide a sound base to measure and analyze the quality of the software thereby analyzing the success of the software development. Software complexity puts its all together as this one metric directly affects the reliability of the software. Even though enough stress is laid on the importance of the software metrics, they still need to grow to meet the required standards. Few metrics that are helping to some extent to decide the reliability of the software systems are included in the Halstead software science.

Halstead software science [35] is a comprehensive set of metrics that allow to measure and analyze different aspects of a program. Halstead proposed the concept of measuring the complexity based on operands and operators in the program to estimate the workload on the programmers. This complexity metric implicitly measures the computational complexity of a given module as a separate single entity.

## 2.4.2 McCabe Cyclomatic Complexity

The cyclomatic complexity as proposed by McCabe is considered as a magic number and very useful in measuring the structural complexity of a computer program.

McCabe Cyclomatic Complexity measures the number of linearly independent present paths in a program and gives cyclomatic complexity represented by a natural number [34], [35]. To measure the McCabe Cyclomatic Complexity, the program must execute according to the decision and control statements such as loops and if/else blocks.

It counts the number of control and decision statements and leaves out the individual chunks of code. The functions with cyclomatic complexity over 10 are very difficult to maintain because of the extra overhead of reliance on branches, switch/cases, or loops.

## 2.5 Online Skill Evaluation Platforms

Open-source code contributions contain a large amount of technical skill information about software engineers. Most of the recruiters refer to business-oriented social networks such as LinkedIn [8], where the profiles are commonly self-authored by the individuals. As much as they are used today, recruiters leave out the aspect of exaggeration and omission of certain skills by the individuals. So following the need, a new trend for developer-oriented sites to measure the skill set of the software engineers is on the rise. For example, platforms such as GitHub [12], Upwork [36], HackerRank [37], and Coderwall [38] help developers to build their profile indicating the qualitative aspects of their technical skills and knowledge.

### 2.5.1 GitHub's Interface

GitHub [12] serves as a platform that reflects the individual's technical skills in contrast to self-authored resumes or online portfolios, which may be an inaccurate display of one's skill set. The user-generated activity builds up an individual's profile thus making GitHub an up-to-date source of information about the developer's exhibited technical skills. Information about a developer's activities on GitHub, such as contributions through pull or merge requests, comments, popular repositories and number of followers, etc. could potentially be used as an indication for measuring technical skills.

Though all of these traits can easily be tracked down on GitHub, the changes made to code repositories are a bit tricky to track down, as it clearly mentions the user of the project has contributed or forked but no direct information about the quality of contribution is provided.

All these challenges makes it difficult to track developers' profile to directly assess their



technical skill set. One has to work manually to get an accurate picture of an individual's technical skill. No direct signals are prevalent to indicate the quality aspects of the individual such as how much experience a developer has in a specific programming language or what other languages they are experienced in, their code quality, and stability. Basic signals provided by the GitHub interface are still helpful enough to derive our own signals to quantify the qualitative aspects mentioned above.

## 2.5.2 Hacker Rank

Focusing on the skill centric hiring process, HackerRank [37] introduces skill-based evaluations [39] for the success required in a technical role. It is a standard platform for assessing the developer's skills helping managers and recruiters to objectively evaluate talent at every step of the recruiting process. The skill assessment might be for universal skills such as problem-solving and code quality or even for specific skills such as expertise in Angular, Micro-services, CSS, etc.

HackerRank maintains a skill directory [39], comprising 55+ individual skills. All these skills can be tested as per the required level by selecting the question from the HackerRank sorted question library. The results help recruiters and managers to identify the skill set and the relevant questions they need for a particular role.

The challenge here is that organization's inclination towards the HackerRank and similar tests will discourage candidates with extensive practical experience in much more challenging environments as few questions from a sorted library will not justify their experiences. Also, since the same questions are being used across all assessment, this might also result as a hack by sharing the solutions of questions over the internet, thus showing that this is not the perfect solution.

## 2.5.3 Waydev

Feedback is one of the fundamental inputs for engineers to help them improve their skills, deepen their knowledge, and bring the best out of them. The understanding of the importance of feedback put forward the need for a tool to highlight their task timeline.

Waydev [40] tracks developer's technical work. It is based on a new agile data-driven method of tracking programmers' through their git activity. It depends exactly on what is needed depending on the specifics of the job and the responsibilities of the engineer. Waydev provides a paid product developer summary [40] highlighting the work patterns and progress over time. The summary includes core code metrics, a breakdown of developer's commit risk, and work focus along with the view of the timeline of their commits. All this information helps to understand the shape of the work done by a developer and

the areas that need to be improved.

## 2.6 Code Analysis Tools

The ever-changing nature of software systems puts a lot of pressure on the development teams. The developers are ought to make sure that quality assurance is done, coding standards are met leaving no room for a single mistake. Thus, developers tend to use static code analysis tools to ensure the code quality.

Static code analysis tools are mainly collections of algorithms and techniques to analyze and debug the source code to identify any vulnerabilities that might lead to production errors and poor coding practices. This can be done manually through code reviews, but automated tools are more effective and time-saving. Static code analysis tools are based on the standard compliance or set of rules such as MISRA C and MISRA C++ [41].

### 2.6.1 SonarQube

Software quality measurement is a quantitative process that depends on various aspects of the software and can be rated along different dimensions.

Developers need to continuously evaluate the code quality to assure a highly reliable end product. SonarQube [42] helps developers to write safer and cleaner code. It is an open-source software that performs a static code analysis to detect bugs, code smells, and security vulnerabilities in source code with support to 27 programming languages. SonarQube continuously inspects the code, from small styling details to critical design errors, protecting the application, and guiding the team. Thus, helping development teams to increase code reliability, software security, and development velocity.

### 2.6.2 Lint

Lint or linter [43] is a static code scanner that identifies, suggests, and corrects the wrong or risky code sections present in the project. It identifies the poorly structured code that is going to pose a threat to the maintenance, reliability, and efficiency of the application. To cite an example, if an XML file contains unused namespaces or a Java source code file with unused imports which only occupy space and require unnecessary processing, deprecated APIs calls that are not supported by the target versions of the project, might lead to code failing to run. Lint helps developers to clean all these issues and improves the overall quality of the project's codebase and accelerates development velocity.

## 2.7 Summary

In the volatile world of software systems, it is very important to ensure the quality of the end product, which depends on the caliber of the developing and testing team. Thus, it is very crucial to quantitatively analyze the technical skills of a software engineer. Although a lot of work has already been done in this respect, there is room for improvement.

Briefly introducing the history and background of the software metrics relevant to this research, we further discussed the limitations of the traditional software estimation methods for measuring the technical skills of a software engineer.

## 3 Design and Methods

In this chapter, a detailed description of the design decisions made for the research is provided. This section builds foundations for every iterative experiment performed to extract qualitative signals. These qualitative signals collectively form the backbone for developing a data model. The purpose of the data model is to create metrics, for assessing technical skills according to an expert’s viewpoint (in this research, the author himself). Thus, an opinionated view is established to provide reasonable quantification of the qualitative aspects of every extracted signal. To ensure the rank ordering of software engineers based on the conceptualized data model, a grading method is designed to relatively measure technical skills.

### 3.1 Creation of Data Model using GitHub Signals

GitHub [12] is an open-source, Source Code Management (SCM) platform where developers or programmers can upload the source code of their software project and work collaboratively to improve it. As of 2018 [44], GitHub hosted 100 million projects maintained by 31 million developers from nearly every country in the world. And one of the major reasons behind the success of GitHub is the fact that it is based on git, a popular open-source Version Control System (VCS). GitHub also provides easy integrations with third-party tools for code review automation, continuous integration and delivery, and task management.

VCS such as git, allows developers to make constant changes to the software code, potentially to fix bugs or release newer versions. VCS records changes to the source code files present in the project repository along with additional audit information such as the name of the author or the time of the change or even additional comments to describe the changes. These changes to the code base are termed as ‘**contributions**’ in the open-source software community. Almost every OSS project on GitHub uses Forking Workflow, which allows every contributor to make a copy of the official project repository and make changes in their personal forked version of the codebase. After completing the changes, the contributor opens a pull request from this forked personal copy to the

original project repository. And only the maintainers of the OSS project are allowed to approve and merge the pull requests into the official project repository after carefully testing and reviewing the code developed by these OSS contributors.

OSS is becoming more popular among the community of programmers. OSS ecosystems have been performing extensive research in the field of software development and have produced successful software projects such as Linux [45], Spring Boot [46], Tensorflow [47], and many others. Thus, contributions to open source projects allow us to infer technical skill information about the developers. As the OSS community prefers GitHub very often, it becomes easy to extract traces representing the qualitative aspects of an individual's technical skills. Such signals could be used to measure technical skills while hiring a potential candidate. Among the online community of developers, there are even claims [48] that

*"GitHub is the new Resume of a developer."*

The work of Amor et al. [49] is particularly significant as it encourages the use of activity signals for effort estimation, especially in the areas of Free and Open Source Software (FOSS). These signals can be used to quantitatively measure the qualitative attributes. For example, if code stability is a qualitative attribute to measure the technical skill of a software engineer then the quantitative metrics would include qualitative signals such as commit frequency, code churn, etc.

Despite the trending popularity of GitHub for open-source softwares, researchers have made claims about the lack of appropriate and efficient research on the GitHub data to prove or make any solid point about signals and their usage for technical skill assessment.

Kalliamvakou et al. [50] identified thirteen perils each representing characteristics of the data that can be retrieved from GitHub. Although these perils question the validity of software engineering researches on OSS projects hosted on GitHub, the study of quantification of qualitative aspects of software development using GitHub signals have continued to make great strides. The research work of Kalliamvakou et al. [50] had been the major source of guidance while configuring our research platform.

### **3.1.1 GitHub API**

Due to the open access to the huge datasets of OSS projects, GitHub has become the target choice for various research efforts in the field of software development [51].

In our research, we used the GitHub rest API v3 [52] to mine GitHub data of OSS projects and extract signals which are effective in measuring the technical skills of a developer. GitHub provides a list of APIs such as Issues, Migrations, Pull Requests, Branches, etc to access data of all activities or events occurring in an OSS project. These

APIs expose information about developers and their activities on project repositories and their characteristics. In our initial research, we observed that most of the signals provided by GitHub endpoints are not useful for quantifying technical skills. For example, user information including email, location, bio, or URLs to the avatar, hooks, tags, etc. do not provide any readily useful signal to measure the technical skills of a software engineer.

Thus, initially, we experimented with very simple and direct signals obtained from the GitHub API that could help us quantify the technical skills of software engineers. And in every successive experiment, we tried to evolve our data model by constructing more refined signals.

### 3.1.2 Data Preparation

REST APIs provided by GitHub contain endpoints for all git related resources such as repositories, contributions, pull requests, issues, comments. These APIs have open access to up to 500 requests per hour per IP address. Also, to be able to extract more data through GitHub APIs, we need to authenticate our requests by generating an access token in the 'Developer Settings' section on the GitHub profile page. With this access token, GitHub allows access to its APIs to up to 5000 requests per hour. To build a data model by extracting qualitative signals through GitHub, we first set up our research platform and defined the scope for measuring the technical skills of a software engineer.

Java, an object-oriented programming language has become a backbone for billions of devices and applications. According to the internet [53], about 3 billion mobile phone devices, 125 million TV sets, and each Blu-Ray player are developed using Java technology. Since Java is at the top in the rankings of the world's most popular programming languages [54], we decided to extract qualitative signals to measure technical skills of a software engineer in Java technology.

After careful study of all thirteen perils and their recommendations to avoid them [50], we decided to analyze a very popular, open-source [55] Spring Boot project in our research. Spring Boot is an open-source Java-based framework used for building web and enterprise applications with 740 active contributors and more than 50,000 stars [46] on GitHub.

To address the research objectives previously outlined, we tried to develop a data model for technical skill assessment of all contributors of the Spring Boot project. Using GitHub APIs, we extracted qualitative signals from the '2.2.x' version [56] of Spring Boot project. Every version of the Spring Boot project is maintained as a branch on GitHub. We also defined the scope of analysis for the period of 2 years from June 18, 2018, to June 19, 2020.

In our research, to prevent bias and hide the developers' identity for confidentiality pur-

poses, we used pseudonyms for developers contributing to the 2.2.x version of the Spring Boot project on GitHub. We developed python scripts to extract data using GitHub APIs and stored the preprocessed data in CSV files for every qualitative signal considered during the experiment. The table 3.1 shows the details of the research platform and the scope of our analysis.

<b>Organization</b>	Spring Projects
<b>Project Repository</b>	Spring Boot
<b>Release Version</b>	2.2.x
<b>Contributor</b>	A (Pseudonym)
<b>Time Period of Analysis</b>	June 18, 2018 to June 19, 2020
<b>Technology</b>	Java

Table 3.1: Research platform and scope of analysis.

## 3.2 Measuring Technical Skills of a Software Engineer

In this section, every experiment performed to quantify the qualitative aspects of software engineering is discussed. We started off by extracting the most common signal from GitHub and in all successive experiments we targeted to extract more refined and detailed signals which would lead us towards our research objectives. Gradually, all assumptions, implementations, limitations of every experiment performed helped each successive experiment to reform the data model and improve its efficiency towards the same goal of quantitative measurement of technical skills of a software engineer.

### 3.2.1 Using Code Commit Signal

#### 3.2.1.1 Overview

According to Gousios et al. [57], a developer’s contribution to the project can be categorized into code commits, documentation, issue or feature discussion, bug reporting, testing, etc. The study performed by Qiu et al. [58] concludes that a developer with more contributions is likely to have higher quality. The major focus of this study was to identify the correlation between the developers’ quality and their contribution to the project measured through metrics such as the number of commits, code quality or stability, etc. Another study [59] measures developer quality based on the developers’ ability to avoid bug introducing commits.

#### What is Code Commit?

In VSCs such as git, engineers write their changes to the software by committing the code change in the project repository. Every commit [60] contains information such as

what part of the software code has changed, the author and the time of change, and a message that briefly describe the change.

In this research, we started our experiments by analyzing the engineer's code committing behavior through a Code Commit Signal. We calculated the total number of commits performed by an engineer on the 2.2.x version of the Spring Boot repository on GitHub. This signal was extracted for a period of 2 years from June 18, 2018, to June 19, 2020. We only considered commits that are successfully merged in the 2.2.x version of Spring Boot project to construct our first metric for measuring the technical skills of a software engineer.

### **3.2.1.2 Implementation**

Code Commit is a direct signal provided by the GitHub API to extract all commits on the project repository.

#### **Implementation Strategy:**

- (1) GitHub's Repo Commit's API [61] allows us to list, view, and compare commits in a repository. This API takes branch hash (SHA) as a querystring parameter to extract commit information on a specific branch (2.2.x) of Spring Boot project.
- (2) In this experiment we used GitHub's List Commits API [61] to extract all commits performed by a contributor 'A' (pseudonym).

### **3.2.1.3 Result**

- In this experiment, we observed that the contributor 'A' has performed a total of **2,508** commits in the 2.2.x version of Spring Boot project.
- Figure 3.1 shows the variation in the commit behaviour of 'A' over a span of 2 years. It has been observed that the total number of commits per month by 'A' decreased over a period of 2 years.

### **3.2.1.4 Discussion**

After analyzing our results, we discovered various limitations of this GitHub signal:

- Although the code commit signal was easy to extract, it did not provide any insights regarding the actual value delivered to the project through each commit. A commit can be trivial for example, fixing a bug by adding a null check or it could be even more complex such as introducing major security-related feature in the software.
- A commit is an arbitrary change that represents nothing more than a save point. Although we have considered commits that have been merged into the 2.2.x version



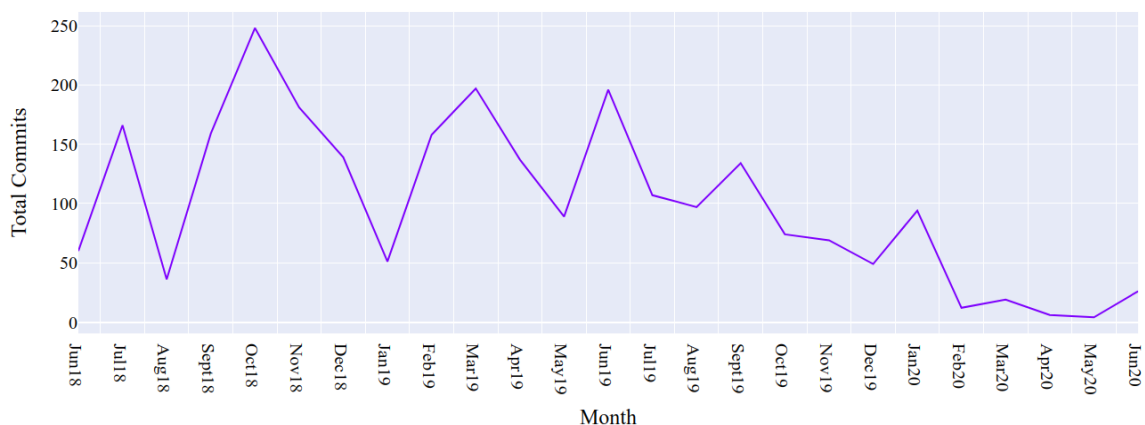


Figure 3.1: Variation in the total number of commits performed by ‘A’ per month

of the Spring Boot repository, we could not quantify the productivity or the value delivered by those commits. We could not quantify the purpose and impact of every commit. For example, if the commit was merged as a part of code refactoring to resolve bugs or introduction of new features by adding new code. It is highly contextual to score 100 small commits more in comparison with 10 big commits having almost 80% of the source code refactored.

- Also, in the organizations where commits are merged without code review by an expert, there exist potential threats that question the reliability of the total code commit signal. For example, developers might make a commit every time they make the smallest possible change on the single line of code.
- Since commit count metric is a mere indication of activity, it is highly unreliable to only depend on this signal for developing a data model that is capable of measuring technical skills of a software engineer. Thus, in the next experiment, we tried to improve our existing data model to capture signals that represent a more reliable committing behavior of an engineer.

## 3.2.2 Using Code Change Signal

### 3.2.2.1 Overview

Lines of Code was one of the first widely used metrics to estimate the size of the software and the productivity of programmers [62]. Lines of Code (LOC) is one important parameter used to measure the quality of the code developed by any software engineer. While writing the code, one cannot claim which part of it is more important than the other. All lines of code hold equal importance for the overall performance of the project. All software engineers must have one or more than one experience of a situation where a single faulty line of code took their hours of debugging. Debugging in massive frameworks is a

real challenge. In such scenarios, changing a single line of code can leave a huge impact. According to Conte et al. [63],

*“a line of code is any line of program text which is not a blank or comment line, irrespective of the number of statements or fragments of statements on the line.”*

For example, all lines of source code that contain program headers, declarations, and executable and non-executable statements.

As explained in [64], the COCOMO model developed by Boehm used Source Lines Of Code (SLOC) metric to size up a software system. This algorithmic model classified the software system into three categories; Organic, Semidetached, and Embedded.

While developing a software system, developer tends to write comments for future references and to maintain the readability of the code sometimes blank lines are also left. Thus, while counting the LOC metric, all comments and blank lines are also counted. Thus, to effectively size up the software, we need to only count the lines of code with the actual programming logic, leaving all the redundant parts. Benedikt et al. [62] introduced a Redundancy-free Source Lines of Code metric that does not include redundant code, test-code since it is not the part of the product deliverable, and also dead-code which is not executed anytime in the product life cycle. To obtain a measurable productivity benchmark for software development projects, Benedikt et al combined effective lines of code with the total effort required and the defect rate metric in the outcome.

In this experiment, we extracted a code change signal for every commit performed by the developer on the Spring Boot project. Using this GitHub signal, we attempted to estimate the total number of lines of code committed by a software engineer. For the purpose of our study, we tracked two types of changes in the code commit signal extracted from GitHub, namely ‘code added’ representing introduction of new code or modifications in the existing code and ‘code deleted’ representing deletion or modifications in the existing code.

### **3.2.2.2 Implementation**

Code Change is a direct signal provided by the GitHub API to capture total number of LOC changed (including both, lines of code added or deleted) in every commit.

#### **Implementation Strategy:**

- (1) In this experiment, we first used GitHub’s List Commits API [61] that returns a unique identifier (SHA) for every commit along with its basic information.
- (2) After extracting each commit performed by developer ‘A’ (pseudonym), We used

GitHub’s Get a Commit API [65] to extract detailed information of every single commit by providing a reference (commit’s SHA) extracted in the step 2.

- (3) To calculate the total number of LOC changed, We iterated through all files present in a single commit and extracted the code change signal from ‘**additions**’, ‘**deletions**’, and ‘**changes**’ keys present in the response of API [65].
- (4) Using file’s extension we calculated the code change signal for all technologies in which developer has contributed. For example: If total number of lines of code changed is 20 in the file **main.java** then we associate this code change value (20) with Java technology (since the file’s extension was **.java**).

### 3.2.2.3 Result

- In this experiment, we observed that developer ‘A’ has changed a total of **113,789** lines of code in Java technology on the 2.2.x version of Spring Boot project. This GitHub signal included lines of code that were newly added, deleted, or modified by ‘A’ over the period of 2 years. GitHub represents modification to a line of code as a combined activity of deleting the existing line of code and then adding the modified line of code.
- Figure 3.2 shows the code change signal extracted from GithHub representing the contribution of ‘A’ in all technologies of Spring Boot project. Almost 80% of the total number of lines of code changed by ‘A’ is in Java technology.
- According to the figure 3.2, 10% of the code change signal represents A’s contribution in XML technology. It has also been observed that ‘A’ has a significant contribution in the documentation of the Spring Boot project.

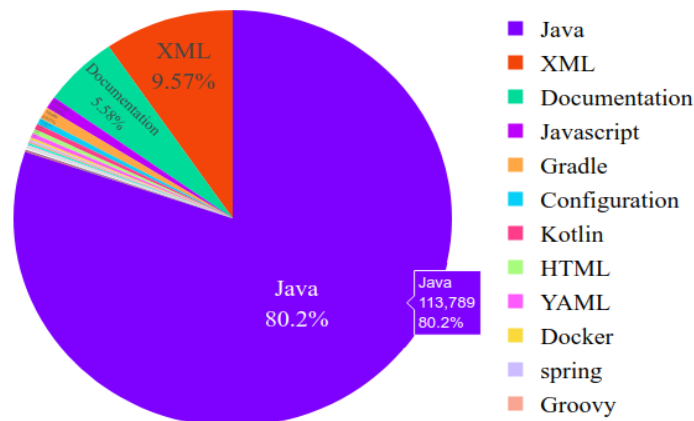
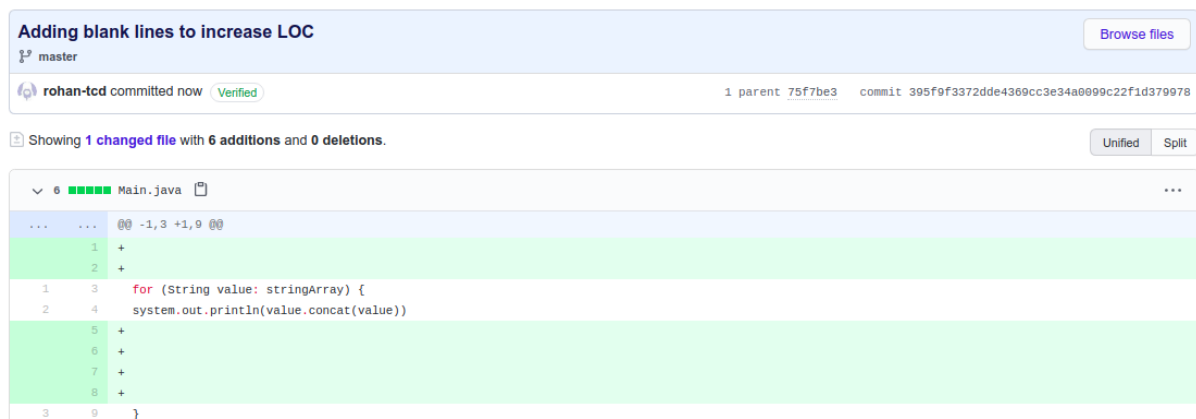


Figure 3.2: Total Number of LOC changed by ‘A’

### 3.2.2.4 Discussion

Although this signal was directly extracted from the GitHub API, it has following limitations concerning its applicability:

- Code Change signal did not provide any options to count the effective lines of code. Since there is no clear agreement for what lines of code are to be considered and whatnot, organizations and industries need to maintain a standard for defining a SLOC metric. For example, while counting the number of lines, if we are excluding the comments for effective SLOC, but doing so we are promoting developers to write code without proper documentation. This is a very poor practice as it will greatly increase the efforts and cost for the maintenance of such software projects with no documentation. Also as shown in figure 3.3, this GitHub signal did not eliminate extra blank lines while calculating the total LOC changed.



```
Adding blank lines to increase LOC
master
rohan-tcd committed now Verified
1 parent 75f7be3 commit 395f9f3372dde4369cc3e34a0099c22f1d379978
Showing 1 changed file with 6 additions and 0 deletions.
Main.java
@@ -1,3 +1,9 @@
1 +
2 +
3 for (String value: stringArray) {
4     system.out.println(value.concat(value))
5 +
6 +
7 +
8 +
9 }
```

Figure 3.3: Inclusion of blank lines in the code change signal extracted from GitHub

- Also in this experiment, while counting the number of lines, we did not differentiate between a complex line of code and an easy line of code. The code content representing the implementation logic should also have weightage when the whole play of software performance relies on factors such as time or space complexity. All developers understand that the same logic can be implemented in various ways and there is a huge difference in the number of lines code.
- We also observed that this metric leaves out the complexity, reliability, and quality of the code. LOC metric cannot relate to the efficiency of the code. Larger code size does not always imply hard or more coding efforts. It might be a less experienced developer who could not utilize all the libraries of a programming language or a programmer developed a lengthy code just to increase the LOC metric. For example, two programs with the same LOC may have two different logics implemented for different purposes and with varying complexity. Also, an engineer could write a lot of sequential statements instead of making use of loops to boost the LOC metric

and thereby reducing the code quality.

- While comparing the LOC metric for different languages to measure the technical skills of a software engineer, the number of lines of code can be entirely misleading. For example, developer ‘A’ writes 1000 LOC, and developer ‘B’ writes 100 LOC in a week. In this scenario, it is hard to compare the productivity of two developers because the two software systems could be entirely different. The same functionality in one language can be more verbose than others. For example, while developing a simple command-line input functionality, the number of lines of code varies significantly in java as compared to python language.
- We also realized that with this GitHub signal, it is hard to distinguish between duplicate code (copying code written by someone else) and new code. Although understanding the existing software code is a challenging task and requires debugging skills for a new or even an experienced engineer, whether to credit such a line of code remains contextual.

### 3.2.3 Using Library Usage Signal

#### 3.2.3.1 Overview

A library is simply a collection of pre-written functions developed as per the supporting language and provides well-defined interfaces by which these functions can be used in other computer programs. Libraries help us achieve one of the most important design principles; *‘Don’t reinvent the wheel’* thereby allowing us to reuse the already developed code.

Joshua Bloch, in his book **Effective Java 3rd Edition** [66], advises every software engineer to use the existing APIs rather than re-writing the code. Inbuilt or external libraries save us a lot of time and effort to develop software systems. We can utilize the saved effort and time in implementing other core business operations.

An important trait of a good software engineer is a wide knowledge of APIs. An Experienced engineer knows how the libraries have been designed to make the most out of it. They use internal or external third-party libraries to develop the software systems, thus saving a lot of time and effort.

For example, skilled software engineers in Java technology can save a lot of implementation time and efforts if they are aware of the existing features of Java libraries including JDK and third-party libraries. For example, knowledge about the **java.lang** package that contains all basic classes including but not limited to String, Double, Math, etc. defined in Java. Similarly, Java collections and data structures are available in **java.util** package. External Java Libraries such as **JUnit**, **Mokito**, **AssertJ** are widely used to

test enterprise java applications, **Log4j** used for abstracting other logging frameworks can all complement the technical skills of a developer. A new developer might not be aware of all such features and spends most of the time and efforts in rewriting the code which might not be as efficient as these well-tested libraries. Although understanding the internal working of every library is challenging and time-consuming, with the knowledge of such libraries an engineer can easily develop the business requirement in a short period. Thus, a skilled programmer is expected to be aware of them.

Our previous experiment illustrated several limitations with the code change metric as the context of measuring LOC was not clearly defined. Thus, in this experiment, to improve our data model, we tried to extend the LOC metric with a specific context. We tried to quantify the usage of an inbuilt or external library by a software engineer. Our focus was to identify how often an engineer uses any library in their work. With this, we could easily assess engineer's proficiency with specific libraries.

### 3.2.3.2 Implementation

Library Usage is an indirect signal that has to be derived using different GitHub APIs and tools. For this experiment, we used Git Blame [67] tool to programmatically identify all lines of code that have been changed in every Java source code file of a given commit. We also developed custom Java source code parsers which were capable of identifying **method calls, name of the calling object, reference of the object (Class name) and the Package or Library to which it belongs** in a given line of code.

#### Implementation Strategy:

- (1) In this experiment, we first used GitHub's List Commits API [61] to extract unique identifier (SHA) for every commit.
- (2) Using this unique identifier (SHA) with Git Blame tool, we programmatically identified every line of code that was changed in the Java source code file.
- (3) We parsed every changed line of code to identify method calls, the calling object, its reference, and the package or library to which it belongs.
- (4) With this approach, we counted each invocation of a method belonging to a particular library.
- (5) After iterating through all Java source code files present in every commit performed by 'A' on the 2.2.x version of Spring Boot project, we extracted the library usage signal by calculating the total number of library method invocations performed by developer 'A' in Java technology.

Figure 3.4 indicates the steps involved in extracting library usage signal from every Java

source code file present in the commit:

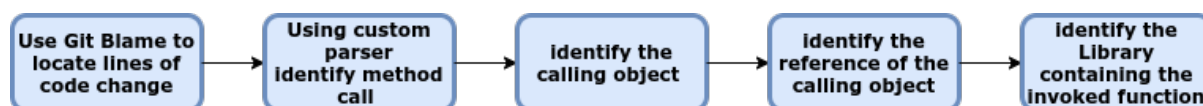


Figure 3.4: Steps involved in measuring Library Usage Signal

A detailed process of calculating Library Usage Signal is outlined in the Appendix with indicative examples.

### 3.2.3.3 Result

- Sunburst chart in figure 3.5 visualizes the library usage count across all contributions of ‘A’ to the 2.2.x of version Spring Boot project over a period of 2 years. ‘A’ has performed a total of **1,310** method invocations for both inbuilt and external third party libraries.
- We observed that **AssertJ** is the most frequently used library by ‘A’ with the total invocation count of 457 in a period of 2 years. The second most used library by ‘A’ is **org.springframework** with invocation count of 331. Also, A used **util**, one of the Java’s core library 201 times.
- With the help of the hierarchical data spanning feature of Sunburst chart, we could also visualize which features (or methods) of the library are most frequently used by a software engineer. For example, according to the figure 3.6 and 3.7 **AssertThat** method is used 453 times out of the total 457 invocations of **AssertJ** library. Also, **Collectors, Collections, List, Arrays**, etc. are the frequently used features from Java **util** library.
- Figure 3.8 shows the overall library usage of ‘A’. According to the figure, ‘A’ has used the AssertJ library for almost 35% of the total library usage signal.

### 3.2.3.4 Discussion

Almost every enterprise-level software system builds on external libraries that provide useful technical features. For example, every technology has library support for common tasks such as logging, testing, collections, data structures, session management, security, etc. Thus, having professional experience with these inbuilt or external libraries is useful for any software engineer. In this experiment, we quantified the inbuilt or external Java library usage of a software engineer.

- The results of this experiment helped us understand which java libraries are most frequently used by the software engineer in their work. With this information, we could also predict their level of expertise with a particular library. For example,

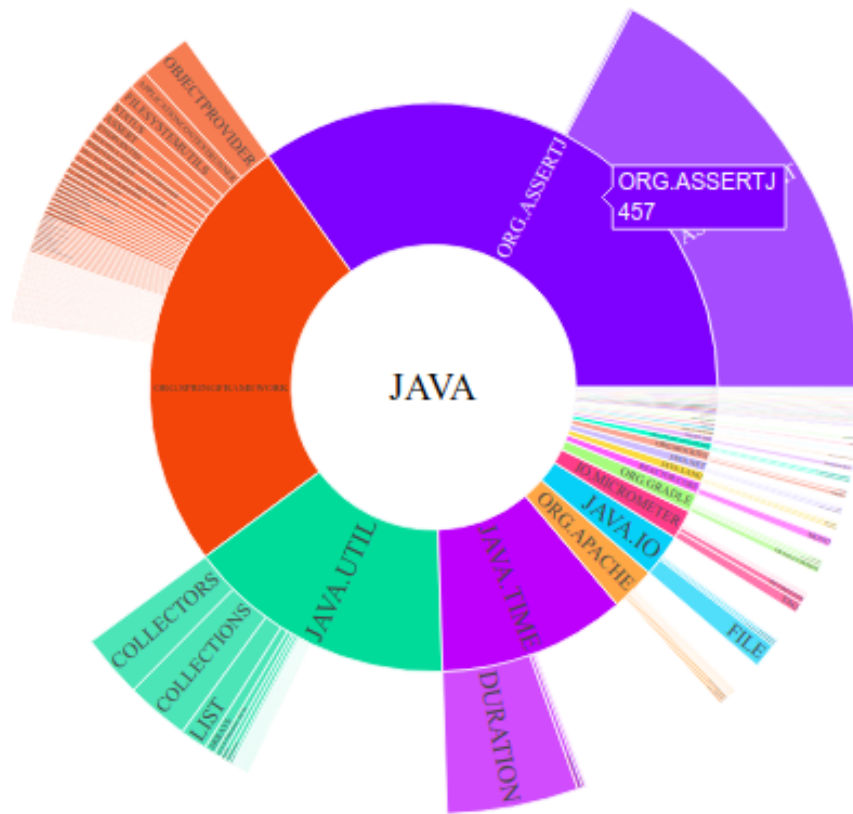


Figure 3.5: Library usage of developer 'A'

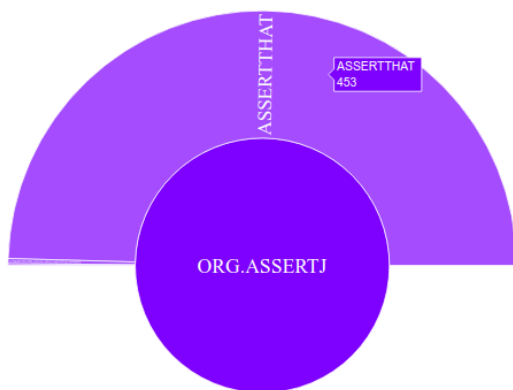


Figure 3.6: Detailed analysis of A's library usage signal for AssertJ library

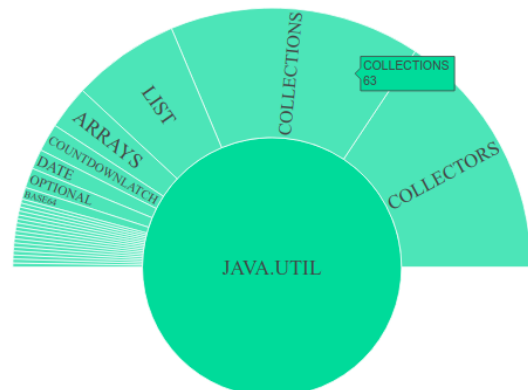


Figure 3.7: Detailed analysis of A's library usage signal for Util library



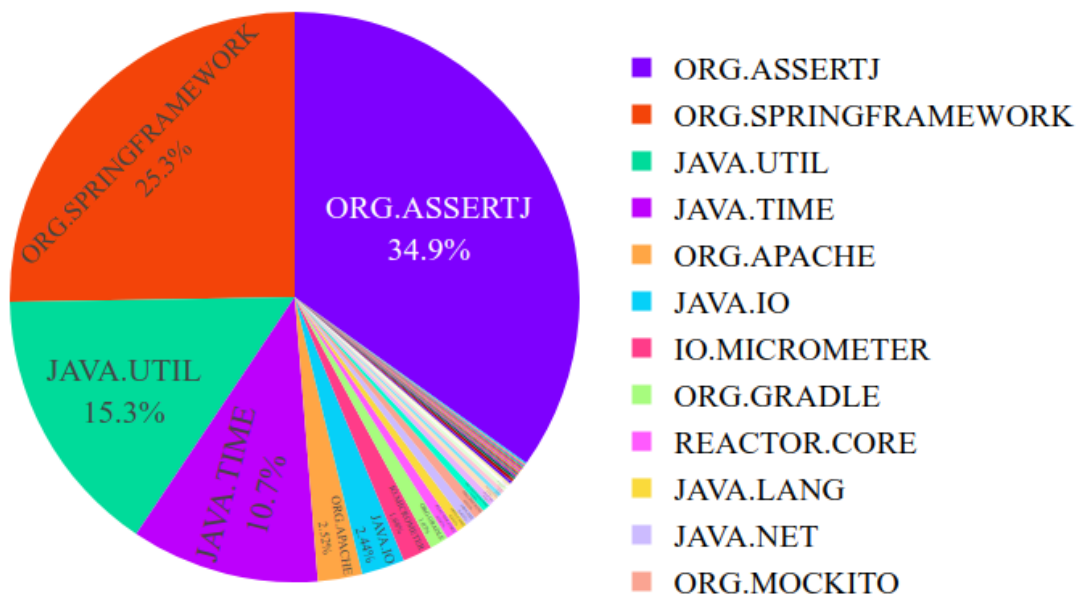


Figure 3.8: Overall library usage by developer ‘A’

according to the figure 3.5 ‘A’ has used the **AssertJ** library for a total of 457 times over a period of 2 years. This frequency count signal gives us a notion of how well experienced ‘A’ is with the AssertJ library. In Java technology, the AssertJ library is majorly used for writing assertions in java test cases. Thus, with this signal, it is safe to assume that the developer ‘A’ is well experienced with the **AssertJ** library majorly used in Software Testing. Similarly, figure 3.5 indicates the total usage count of 201 for Java’s util library which majorly contains APIs for handling data structures in Java programming language.

- In this experiment, we developed custom Java source code parsers based on regular expressions to identify method calls and their libraries. We observed that in certain situations the custom parsers were not efficient in identifying method calls accurately. Also, due to different coding styles in Java language, custom parsers could not identify package names of libraries for invocations in method chaining. For example, in the code snippet given below, methods are invoked in the chained fashion. And in such coding styles, it is not feasible to predict the return type of the intermediate methods that are chained together.

```

1 public static void main(String[] args) {
2     MethodChainingExample obj = new MethodChainingExample();
3     // Output: In method chaining, methods are invoked from left to
4     // right
5     obj.setName("A").setAge(10).introduce();

```

- In the above example, it is difficult to identify the return type of the `setAge` method which would later call the `introduce` method. We need debugging tools to manually identify the return types of internal methods.
- To overcome the limitations of our custom parsers and improve the accuracy of the library usage signal, we could make use of Abstract Syntax Trees (AST) that returns the syntactic structure of source code and encodes syntactic relations between various source code elements. For example, tools such as PMD [68] retrieves all method calls by constructing an AST of the given source code file.
- We could also extend this signal to understand the importance of a software engineer within a project. With the help of library usage signal, we can create a network of developers and their library usage. For example: If ‘A’ develops a package or library that is widely used by other engineers in their work then we could construct a network of libraries where each edge of the library node indicates its usage by other developers. Thus, if the library node has more associations with the other engineers (more number of edges) then this might indicate that its creator is highly valuable to the project.

## 3.2.4 Using Code Churn Signal

### 3.2.4.1 Overview

Software evolution is an interactive process of integrating new functionalities while introducing and resolving existing bugs. Many new integrated features along with the fixed defects often introduce more defects into the system. Even changing a single line of code can raise the risk of new defects in the system. Even after committing some new code into the codebase, it might need to be reverted due to the new defects introduced by it. Thus, software engineers have to be very careful with the changes they make to the software to avoid introduction of new bugs and rework.

#### What is Code Churn?

Code churn represents the rework that the software engineer had to do to resolve bugs introduced by their previous work. Typically, code churn is when an engineer refactors their own code in a very short period (approximately within 2 weeks).

According to Khoshgoftaar et al. [69] Code Churn is defined as

*“the number of lines added to, deleted from, or modified in a source module”*

It is measured as the lines of code that were modified, added, and deleted over a short

period. Although churn is an inevitable part of the software development life cycle, it should be as low as possible. The main purpose of the code churn is to understand the maintenance and quality checks needed for the safe evolution of the software. The system quality can be measured by assessing how easy a change is in the system. All these reasons make it a common belief that modifying a high churn software system is more likely to cause its degradation.

Considering the importance of the effect of modified code on the project, many researchers have researched to understand the code churn signal. According to Graves et al. [70], there is a high rate of defects after the code has been changed as compared to the earlier version of the code. Also, Nagappan et al. [71] found that the higher the modification of the component, higher are the chances of post-release defects in that component. Ohlson et al. [72] developed a model using code churn and other metrics to discover more fault-prone software components that required corrective maintenance. The study was constructed on 8 releases of a legacy software having 130 components. They were able to identify the most fault-prone components with the help of code churn and other metrics that required extreme maintenance. In another qualitative study, the authors [73] discover 13 common reasons for code reverts by manually inspecting 3,144 reverting commits.

Thus, in this experiment, we tried to extract the code churn signal for every developer to understand the quality of their commits on the GitHub project.

### **3.2.4.2 Implementation**

Code churn is an indirect signal that has to be derived using different GitHub APIs and tools. we used Git Blame [67] and Git Log [74] tools to programmatically identify the number of lines of code that have been survived as of June 19, 2020, since its initial commit date.

#### **Implementation Strategy:**

- (1) In this experiment, we first used the GitHub's List commits API to extract the unique identifier (SHA) for every commit.
- (2) Git Blame tool annotates each line in the given source code file with the information such as commit's hash (SHA), name and email of the author from the revision which last modified the line. We used the commit hash obtained in the previous step with the Git Blame tool and programmatically calculated the total number of lines of code that was changed in a given Java source code file by counting the occurrences of commit hash (SHA) showed in the Git Blame revision.
- (3) Using the Git Log tool on a given Java source code file, we identified all unique

identifiers (SHA) of commits performed after the commit under analysis. Using this tool we could easily list unique identifiers (SHA) of all commits by parsing through the commit history of the source file provided by the git log tool.

- (4) In the next step, We checked if the code changes which are part of the commit under analysis are still present in the latest commit revision (dated June 19, 2020) performed on a given Java source code file. If the commit hash (SHA) count (calculated in step 2) does not match with the total count currently present in the latest commit, we marked our commit under analysis for calculating code churn.
- (5) The number of churned or refactored LOC was calculated by taking the difference of the total commit hash (SHA) count of the initial commit (calculated in step 2) with the total count of the same commit hash (SHA) present in the latest commit revision of the source file provided by Git Blame.
- (6) The number of surviving lines of code was also calculated by counting occurrences of the initial commit hash (SHA) (calculated in step 2) present in the latest commit revision of the source file.

To calculate code churn signal of a software engineer we used following formula:

$$CodeChurnRate := \frac{NumberOfLOCDeleted}{NumberOfLOCAdded} * 100 \quad (1)$$

A detailed process of code churn identification is explained in the Appendix with indicative examples.

### 3.2.4.3 Result

- Figure 3.10 indicates the variation in the total number of LOC added in a given month. It also indicates how many LOC are churned (out of the total LOC added) as of June 19, 2020. For example, according to the figure 3.9, ‘A’ added a total **1,505** lines of code in the month of December 2018, and out of these added LOC, **866** lines of code are churned (refactored or deleted) as of June 19, 2020. Thus, according to the formula:

$$CodeChurnRate := \frac{NumberOfLOCDeleted}{NumberOfLOCAdded} * 100 \quad (2)$$

A’s code churn rate for the month of December 2018 can be calculated as:

$$CodeChurnRate = \frac{866}{1,505} * 100 = 57\% \quad (3)$$

- From figure 3.10, we can confirm the code churn rate of **57%** in the month of December 2018. Also, from this figure it is quite evident that the code churn rate of ‘A’ reduced over a period of 2 years.
- Thus, we observed that software engineer ‘A’ has an average code churn rate of **26.19%** across all 2,508 commits.

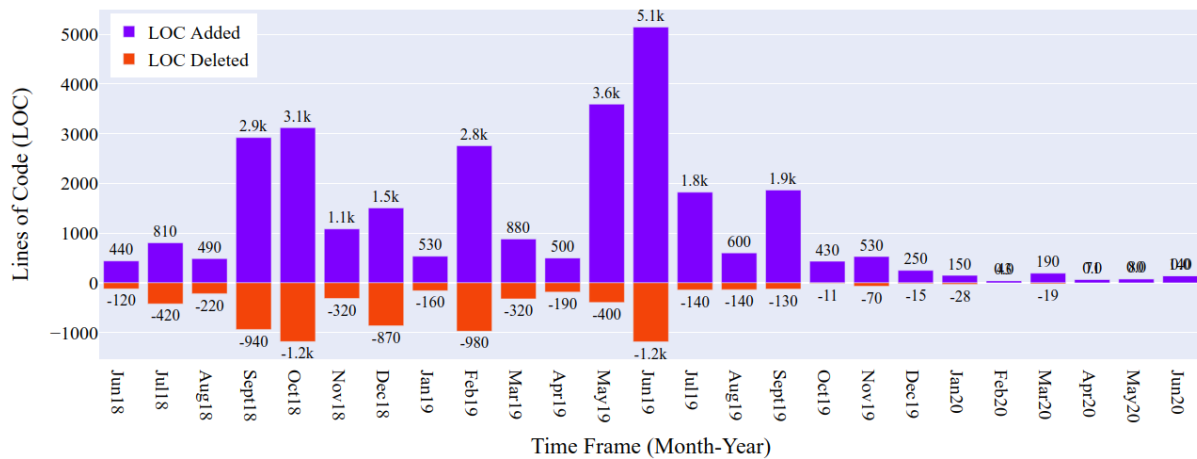


Figure 3.9: Variation in the total number of LOC added by ‘A’ vs. total number of LOC churned (or deleted) as of June 19, 2020

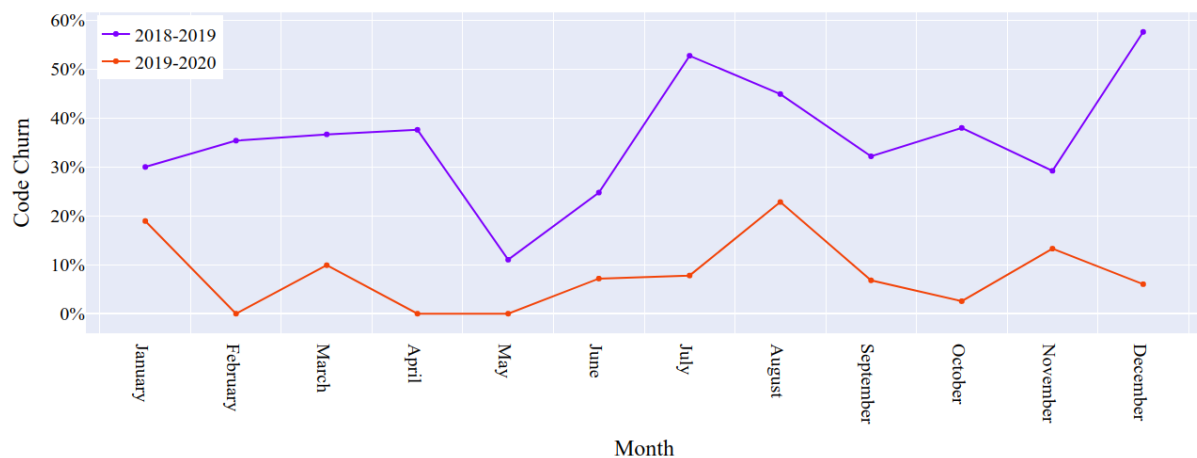


Figure 3.10: Variation in A’s code churn rate signal

### 3.2.4.4 Discussion

After analyzing our results, we realized that the code churn signal is highly contextual because of the following observations:

- Software enhancement is a crucial part of the SDLC, and to some extent, rework is completely a normal thing. Sometimes for challenging projects, development teams have to continuously refine their approaches during the initial system design phase. Also, when the software is completely new in the market, it is expected to have a

high code churn. Also, when an engineer is innovating a new feature or stepping into an exploratory project, high code churn rates are normal. Prototypes, POCs, and redesigns are all required to go through large chunks of code churn. Though it seems to be a lot of rework initially, by the end it justifies the quality of the code. Also, in such fast-paced development environments, the engineer could release out solutions to new innovative ideas quickly.

- Another observation indicated that the code churn can be completely contextual, for example, if the stakeholder is not clear about the requirements and frequently changing their mind, it may lead to high code churn rate and may also have a negative impact on the team morale for extensive rework.
- Also, a very challenging problem demands exploring multiple solutions to reach an optimum implementation, thus resulting in a higher code churn. Though it's completely contextual, as to what is difficult and whatnot sometimes, it might be due to engineers thinking about the problem in a way different than it should actually be understood. Hence demanding a lot of rework.
- On the other hand, sometimes low throughput by an engineer due to boredom or burnout might lead to a low code churn. As engineer is not contributing regularly or focusing on one particular part of code more than the required time, there are high chances that the code churn rate would be lower as compared to other constantly performing engineers.

## 3.2.5 Using Code Stability Signal

### 3.2.5.1 Overview

Due to the evolutionary nature of the software, code stability plays an important role in the measurement of software quality. Software quality can be boosted by producing software free from frequent and substantial changes. Research presented by Menzies et al. [75] targeted avoiding dramatic software changes by using the SEESAW model based on the stochastic stability method. Reuse-based software development [76] has already been researching ideas to create stable software artifacts so that they can be reused with minimum modifications.

In recent researches, software stability has only been measured using architecture-level metrics such as reference points and program-level metrics including the number of lines of code and number of modules. Threm et al. [77] used information-level metrics based on Kolmogorov complexity to measure the stability by comparing different versions of the software. Using information-level metrics authors could monitor the software evolution process by identifying stable or unstable software artifacts.

## What is Code stability?

Code stability signal associates every line of code change with the time elapsed since its initial commit. It analyses how reliable every line of code change is over the period of time.

In the previous experiment, we derived the code churn signal to be incorporated into our data model that is used to measure the technical skill of a software engineer. Extending our research further, in this experiment, we tried to analyze the stability of code commits by the developer. We attempted to create a new metric by crediting every line of code committed by a developer which is stable and remained unchanged over the evolution of the software.

### 3.2.5.2 Implementation

Code Stability is an indirect signal that has to be derived using code churn signal. In this experiment, we used code churn signal obtained in the previous experiment to get the number of lines of code that have been survived as of June 19, 2020, since its initial commit date.

#### Implementation Strategy:

- (1) In this experiment, we measured the total number of surviving lines of code (as of June 19, 2020) from every commit performed in the time frame of 2 years from June 18, 2018, to June 19, 2020.
- (2) We calculated the number of LOC that are survived from all commits by using the strategy as described in section 3.2.4.2.
- (3) In the next step, we credited points to every surviving line of code according to the number of days elapsed between its initial commit date and June 19, 2020.
- (4) Code Stability signal is calculated using given formula 4:

$$\text{CreditPointsForCodeStability} = \text{NumberOfLOCSurvived} * \frac{\text{NumberOfDaysElapsed}}{365} \quad (4)$$

A detailed case study of measuring code stability signal is provided in the Appendix.

### 3.2.5.3 Result

- Figure 3.11 indicates the variation in the total number of LOC added by 'A' in a given month. It also indicates the number of survived LOC (out of the total LOC added) as of June 19, 2020. For example, according to the figure 3.11, 'A' added a

total **1,505** lines of code in the month of December 2018, and out of these added LOC, **639** lines of code are survived (or still present in the project’s codebase) as of June 19, 2020.

- Using formula 4, we credited every survived LOC based on the number of days elapsed between its initial date and the date of analysis (June 19, 2020).
- Figure 3.12 indicates the code stability credit points earned by ‘A’. Thus, ‘A’ earned a total of **26,434.61** code stability credits in a period of 2 years.

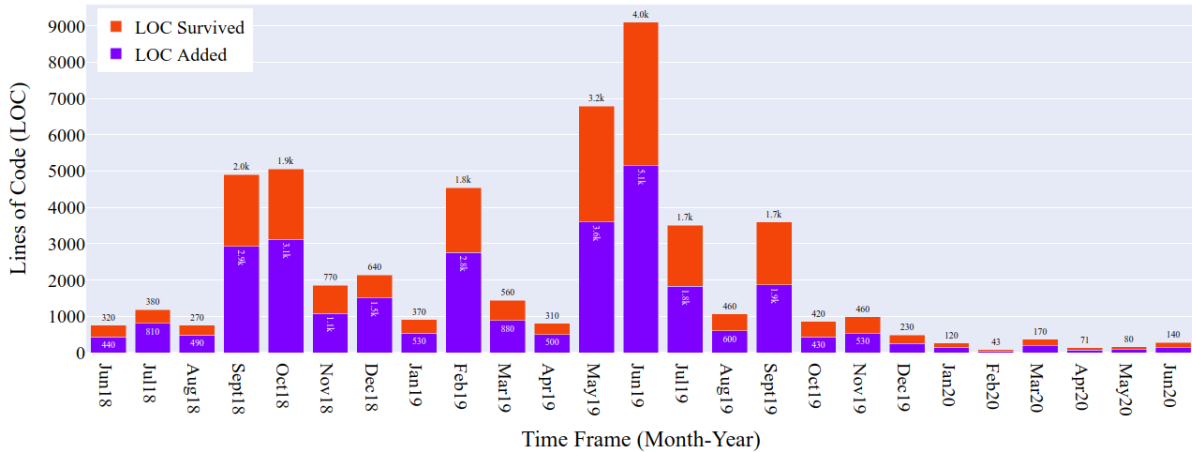


Figure 3.11: Variation in the LOC added by ‘A’ vs. LOC survived as of June 19, 2020

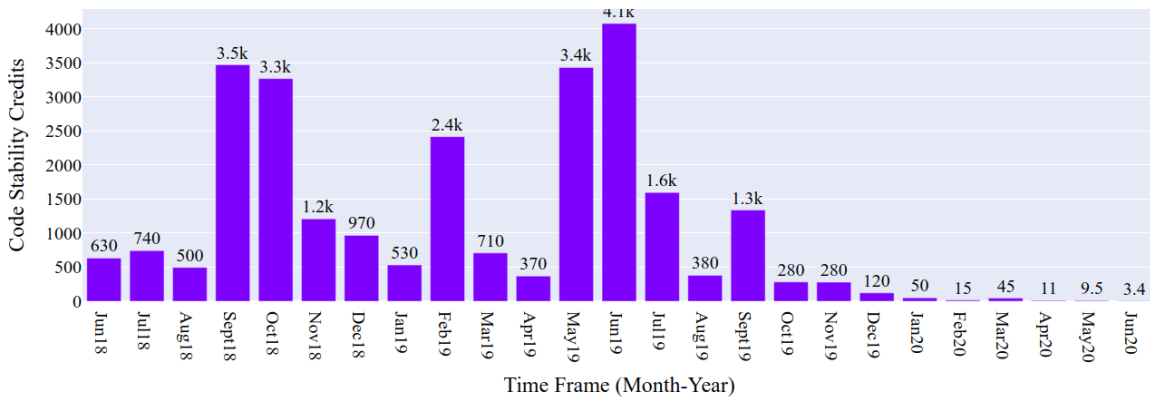


Figure 3.12: Code stability credits earned by ‘A’ every month

### 3.2.5.4 Discussion

- Code Stability signal accurately weighed the importance of every line of code that was changed in a commit. Since it considered the number of days elapsed between its initial commit and day of analysis, stability signal assessed the reliability of every line of code change during software development phase.
- With the credit points assigned to each commit, we could understand the stability of code produced by every developer contributing to the project.



- As discussed in the section 3.2.4.4, code stability signal is also highly contextual and may not be applicable for development environments such as research and innovation where it is expected to have low stability of the code. As developers might be working on an innovation which might require trying different approaches everyday to identify an optimum solution.

### 3.3 Measuring Technical Skills using Data Model

In the chase of an efficient expert system, to quantify the technical skills of a software engineer, we realized peculiar points about the software evolution over the time. We performed several experiments to measure each metric and realized that no single metric in itself is stable enough to measure the technical skills of a software engineer. Each experiment showcased the limitation of the concerned metric and paved the way for another metric to join the regime.

From all experiments, we concluded that the use of these signals can be highly contextual. For example, code churn is the percentage of the count of refactored (modification or deletion) LOC after it was initially committed to the project's codebase. In one context, where an engineer is working on a stable software solution in the banking or finance domain where the code churn should be as low as possible. But in another context, if the research team is trying to innovate a solution, higher code churn is normally expected. In such environments, development teams might be constantly trying different approaches to find the optimum solution thereby introducing high code churn. In this situation, a low code churn rate might indicate negligence in the engineer's effort to innovate.

To develop an expert system as the prime objective of this research, we had to form an opinionated view to set up the context for each signal, so that they can be effectively used to assess technical skills of a software engineer. Thus, for every extracted signal, we provided a context based on our own experience and point of view.

We based our data model on the following assumptions:

*“A higher number of commits suggest higher technical skills.”*

Software engineers with higher commit count might have worked on different issues and contributed frequently to improve software quality and performance.

*“A higher number of lines of code changed suggest higher technical skills.”*

Software Engineers with a high number of LOC changed might have worked on the software issues that required major changes to the codebase. For example, refactoring existing software architecture or workflows.

*“A higher programming library usage indicates higher technical skills.”*

Software Engineers having higher library usage in their work might indicate that they are well-aware of inbuilt or external library features in given technology.

*“A higher code churn indicates lower technical skills.”*

A higher code churn rate might indicate that software engineers could not accomplish their task in the first attempt efficiently. Poor software testing or lack of in-depth analysis might introduce new defects thereby introducing rework.

*“A higher code stability earns a high rating for technical skills.”*

As the code committed by software engineer age, it adds stability to the software. Such a piece of code should have high importance over constantly changing code.

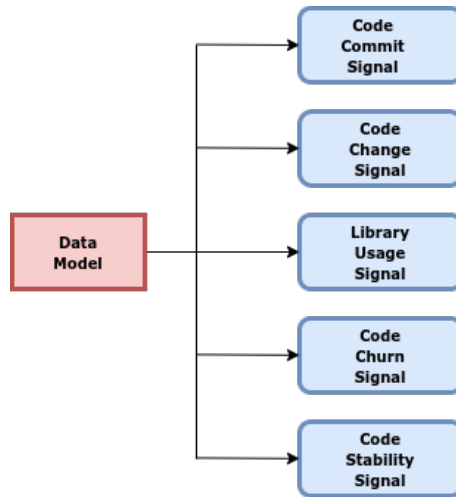


Figure 3.13: Data model to measure technical skills of a software engineer

Since this data model is based on above opinionated view, it might not be a perfect fit for all environments of software development. As discussed above, these views are highly contextual in the light of the above-mentioned assumptions.

### 3.4 Measuring Technical Skills using Manual Evaluation

The primary objective of this research was to measure the technical skills of software engineers by developing a data model. The data model is created using various qualitative signals representing activities carried out by the engineers in the OSS project. The experiments performed for measuring these signals highlighted their contextual nature along with their limitations. Thus, an expert system is created based on the opinionated view of the data model to rank order software engineers according to their technical skills.

According to Ackerman et al.[78], the quality of a software project can be improved greatly if the source code is manually inspected by peer developers instead of the author in particular. Thus, to test the efficiency of this automated expert system, we manually verified its outcomes. Results obtained by the expert system were verified against the knowledge of a human expert (in this research, the author himself). The manual evaluation was based on the knowledge and understanding of the human expert for every signal measured by the expert system. The manual analysis considered the limitations of every experimented signal while analyzing the efficiency of the expert system. Manual analysis majorly focused on how well the limitations of every signal have been covered by the automated system.

Figure 3.14 presents the evaluation framework for manual analysis. This framework lists various criteria used by the human expert while manually analyzing every signal. For example, signals such as code churn was verified to understand the context of the development environment and the purpose of code churn, whereas the code commit signal was manually assessed to determine the value and reliability delivered by every commit. The code stability signal had been verified to determine the stability offered by every code changed in the given development environment. Library usage signal was attested to measure the accuracy of custom parsers for extracting complex method calls. The code change signal was analyzed to differentiate between the use of effective LOC including comments, complex implementation logic and LOC with black or duplicate lines of code. Thus, to quantify the effectiveness of the manual model of analysis, every signal was rewarded points in the range 0-100. The Appendix provides an elaborated study for the manual verification process.

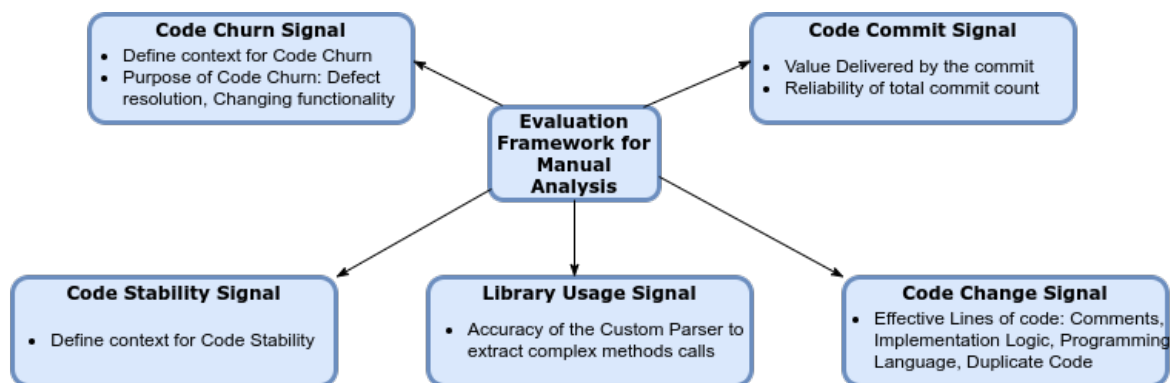


Figure 3.14: Evaluation framework for manual analysis

## 3.5 Ranking Mechanism of Expert System

### 3.5.1 Overview

This section describes the ranking mechanism used by our expert system to quantify the technical skills of a software engineer.

An expert system is defined as an intelligent decision-making system that uses both knowledge and inference procedure (heuristics) to emulate human-level intelligence and expertise.

As discussed in section 3.3, the expert system rank orders software engineers according to the opinionated view of each qualitative signal defined in the data model. To design the inference engine of the expert system, we applied standard mathematical transformation on each signal extracted through GitHub to rank order different software engineers by comparing their technical skills.

As discussed in section 3.3, every experiment performed to extract qualitative signals representing the technical skills of a software engineer helped us develop a data model. Table 3.2 shows the quantification of every qualitative signal extracted from GitHub for software engineer ‘A’ contributing to the 2.2.x version of the Spring Boot project. Each signal represents the technical expertise of a software engineer in Java technology.

<b>Qualitative Signals</b>	<b>Results</b>	<b>Description</b>
Code Commit Signal	2,508	Total Number of Commits
Code Change Signal	113,789	Total Number of LOC Changed (Added + Deleted)
Library Usage Signal	1,310	Total Number of Java Library Invocations
Code Churn Signal	26.19%	Ratio of total LOC deleted to total LOC added
Code Stability Signal	26,434.61	Credit assigned for every stable LOC

Table 3.2: Data model with quantified signals representing technical skills of developer ‘A’

From table 3.2, it is evident that every signal represents technical skills with an unbounded arbitrary number. Also, every signal quantifies technical skills on different measurement standards. For example, the library usage signal represents frequency as an arbitrary number while the code churn rate signal represents the ratio of the total number of LOC deleted to the total number of LOC added as a percentage. Thus, to overcome these challenges we used relative grading system to maintain uniformity in the measurements of all qualitative signals. Using this inference mechanism we could transform all measures to a standard scale.

## 3.5.2 Measuring Technical Skills

Absolute (criteria-referenced) and Relative (norm-referenced) grading are the two most widely used grading schemes in businesses, schools, and universities [79].

### **Absolute Grading:**

In this method, the grades are determined by a fixed standard. In the absolute grading method, a specific range of scores is assigned a specific grade. For example, a student with an annual academic percentage in the range of 90-100% is graded ‘A’ while another student with a percentage in the range of 70-90% is graded ‘B’. Under absolute evaluations, grades do not consider relative performance. Also, there should be a predefined threshold on the scores that need to be graded. Thus, we used the relative grading mechanism to introduce uniformity in all qualitative signals extracted from GitHub.

### **Relative Grading:**

In this method, there is no predefined threshold on the scores obtained by the candidates. Relative grading does not categorize grades based on the predefined ranges of the scores. It defines grade as an expression of how well a particular candidate performed in comparison with another candidate (mostly the topmost candidate) For example, the topmost candidate with the highest score receives full grades, and the rest of the candidates receive relative grades in comparison with the scores obtained by the topmost candidate.

In this research, points obtained for every qualitative signal using Relative Grading system are discussed in the section 4.2.

## 3.6 Expert System Design

### 3.6.1 Creation of Technical Skills Assessment Dashboard

This section describes the implementation of the technical skill assessment dashboard.

Figure 3.15 gives an overview of steps involved in the implementation of an Expert System capable of measuring technical skills of a software engineer.

1. In the first step, we developed python scripts to extract qualitative signals using GitHub Rest API [52].
2. In the next step, we transformed extracted GitHub signals to generate more effective qualitative measures and stored them in a CSV file that acted as a database.
3. For effective technical skill assessment, we developed visualization dashboard for our expert system.

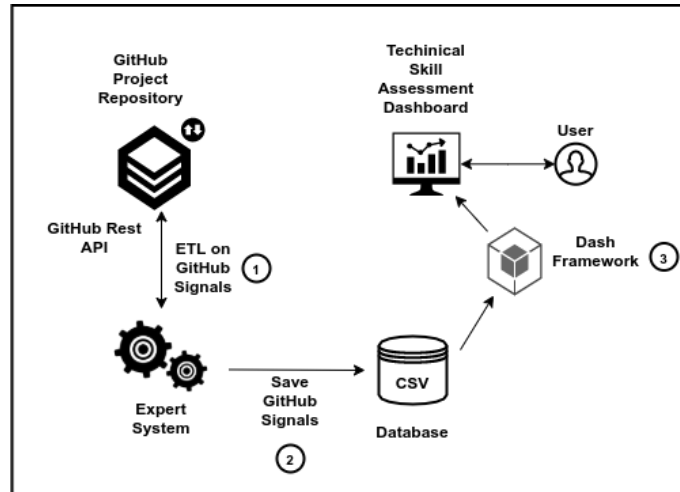


Figure 3.15: Architecture of the expert system

The dashboard is designed and developed using Dash framework [80]. All data visualizations such as Bar chart, Line chart, Sunburst chart, and Pie chart are provided by Plotly [81] library. This Dashboard acts as the front end of our expert system that rank orders software engineers by measuring their technical skills. Dash framework [80] has inbuilt support for Plotly library to integrate and dynamically update the visualizations using callback functions.

This Dashboard contains a drop-down option to select the engineer whose technical skills need to be measured. On selection of the engineer, it gives a detailed analysis of technical skill assessment for every signal extracted from GitHub.

The illustrations of the dashboard, quantifying technical skills of a software engineer ‘A’ are given below:

DISSERTATION  
**SOCIOMETRICS IN SOFTWARE ENGINEERING: MEASURING TECHNICAL SKILLS OF SOFTWARE ENGINEER THROUGH GITHUB**

**SPRING BOOT**

TECHNICAL SKILLS ASSESSMENT DASHBOARD

SELECT EMPLOYEE

A

**1**

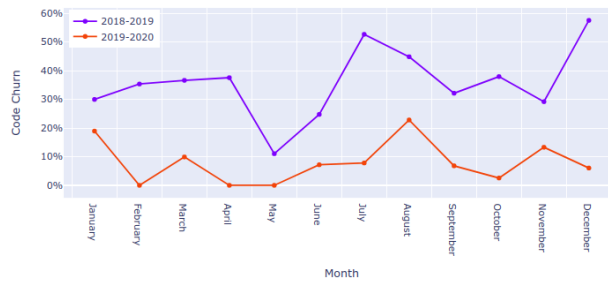
481/500

**A**

SPRING BOOT: 2.2.X

CODE COMMIT SIGNAL: 2508 | CODE COMMIT POINTS: 100  
 CODE CHANGE SIGNAL (JAVA): 113789 | CODE CHANGE POINTS: 100  
 LIBRARY USAGE SIGNAL (JAVA): 1310 | LIBRARY USAGE POINTS: 100  
 CODE CHURN SIGNAL (JAVA): 26.19% | CODE CHURN POINTS: 94  
 CODE STABILITY SIGNAL: (JAVA) 26434.61 | CODE STABILITY POINTS: 87

A's Code Churn Rate in Java (June 2018 - June 2020)



A's Total number of LOC Changed (June 2018 - June 2020)



A's Total LOC Added Vs. Total LOC Churned (Deleted) in Java as of June 19, 2020

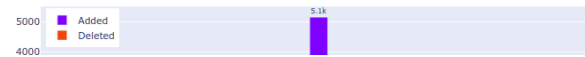
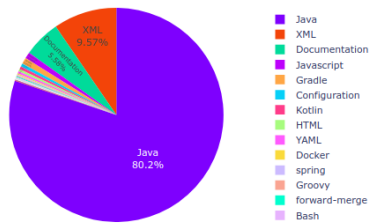
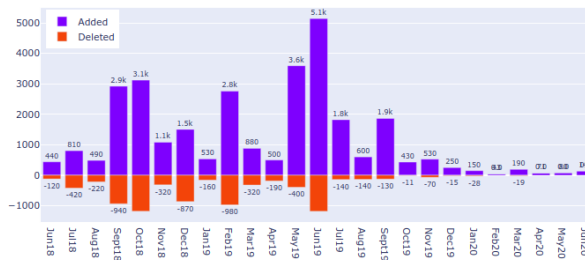


Figure 3.16: Technical skill assessment dashboard 1

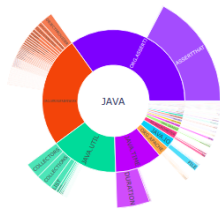
A's Total number of LOC Changed (June 2018 - June 2020)



A's Total LOC Added Vs. Total LOC Churned (Deleted) in Java as of June 19, 2020



A's Library Usage in Java: Detailed Analysis (June 2018 - June 2020)



A's Code Stability Credits in Java (June 2018 - June 2020)

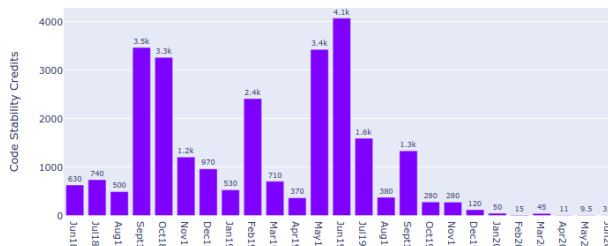
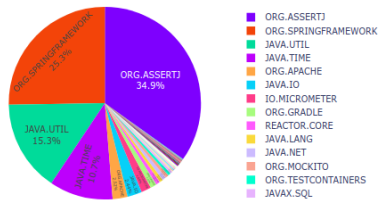
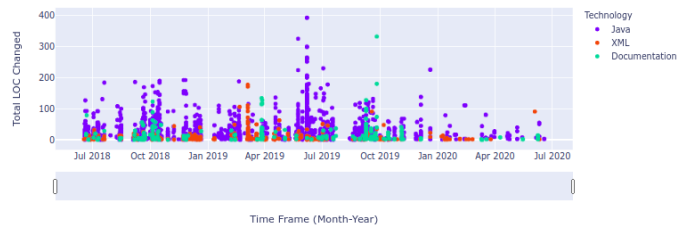


Figure 3.17: Technical skill assessment dashboard 2

A's Overall Library Usage in Java (June 2018 - June 2020)



A's Top 3 Technology Usage Analysis (June 2018 - June 2020)



TECHNICAL SKILLS COMPARISON DASHBOARD: TEAM VIEW

Comparing Code Commit Signal of All Software Engineers (June 2018 - June 2020)

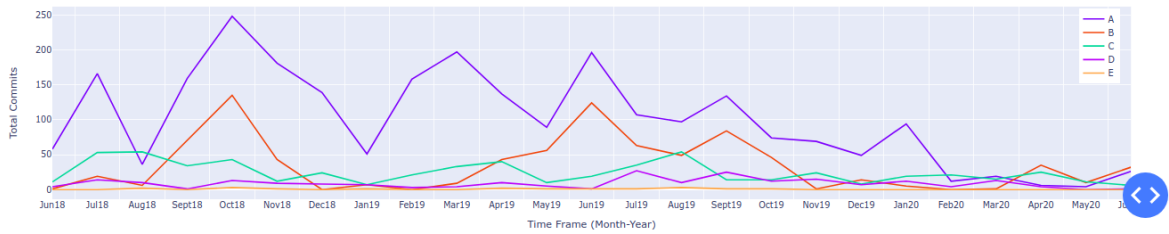


Figure 3.18: Technical skill assessment dashboard 3



## 4 Result and Evaluation

The final phase of the thesis points out the results generated by the proposed system along with its evaluation against the accepted standards. The results provided by the expert system were compared with the results obtained by the manual model of evaluation. Pearson’s correlation was utilized to investigate the strength of association between the results obtained by two models.

### 4.1 Data Collection

To evaluate the data model, GitHub signals representing the activities of 5 software engineers who are contributing to the 2.2.x version of Spring Boot project are extracted. As discussed in section 3.1.2, all qualitative signals have been collected for the time frame of 2 years from June 18, 2018 to June 19, 2020. We developed python scripts to extract these qualitative signals using GitHub REST APIs. The user profile details including the name and email of 5 engineers are concealed to prevent bias and maintain confidentiality.

### 4.2 Software Engineers Ranking by Expert System

Table 4.1 shows the quantified values of all extracted signals which represent the technical skills of software engineers.

<b>Software Engineer</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>
<b>Code Commit Signal</b>	2,508	854	608	219	17
<b>Code Change Signal</b>	113,789	98,709	61,616	6,446	606
<b>Library Usage Signal</b>	1,310	1,194	369	259	24
<b>Code Churn Rate Signal</b>	26.19%	21.56%	27.32%	33.65%	37%
<b>Code Stability Signal</b>	26,434.61	30,352.87	14,122.83	1,503.88	379.515

Table 4.1: Data model representing the quantified values of extracted signals for all software engineers

As discussed in section 3.5, to overcome the challenges in the measurement of these signals on a standard scale, we used the Relative Grading system.

## Relative Grading System

In section 3.3, we discussed the opinionated view to set the context for every qualitative signal used in the data model. These heuristics helped to design the inference engine of the expert system. Thus, with the help of relative grading system, the inference engine could quantify all qualitative signals on a standard scale of measurement. We set the the measurement scale of 0-100 points for every qualitative signal extracted from GitHub.

For example, according to our opinionated view of the code commit signal defined in section 3.3, relative grading system assigns maximum points to the candidate having highest value of code commit signal. With this system, the topmost candidate gets maximum points and rest of the candidates receives relative points in comparison with the score obtained by the topmost candidate. Thus, for the code commit signal, our expert system assigns 100 points to the software engineer 'A'. Since the code commit signal is the highest for 'A', other candidates receive relative points in comparison with the code commit signal of 'A'. Similarly in the case of code churn signal, software engineer with low code churn receives maximum points.

Table 4.2 indicates highest value of every qualitative signal based on the opinionated view of the data model.

<b>Software Engineer</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>
<b>Code Commit Signal</b>	2,508	854	608	219	17
<b>Code Change Signal</b>	113,789	98,709	61,616	6,446	606
<b>Library Usage Signal</b>	1,310	1,194	369	259	24
<b>Code Churn Rate Signal</b>	26.19%	21.56%	27.32%	33.65%	37%
<b>Code Stability Signal</b>	26,434.61	30,352.87	14,122.83	1,503.88	379.515

Table 4.2: Data model representing the highest value of every qualitative signal

Thus, with the help of conceptualised data model and the relative grading system, our expert system measures technical skills of a software engineer on a scale of 1-100 for every qualitative signal extracted from GitHub.

Table 4.4 indicates the technical skill points assigned by expert system for every qualitative signal.

Thus, our expert system could quantify technical skills of software engineers based on these 5 qualitative signals. Table 4.4 shows the total points received by every software engineer out of 500. With the help of this total points metric, the proposed system rank orders every software engineer as per their technical skills represented by the qualitative signals of the data model.

<b>Software Engineer</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>
<b>Code Commit Signal</b>	100	34	24	9	1
<b>Code Change Signal</b>	100	87	54	6	1
<b>Library Usage Signal</b>	100	91	28	20	2
<b>Code Churn Rate Signal</b>	94	100	93	85	80
<b>Code Stability Signal</b>	87	100	47	5	1
<b>Total Points</b>	481	412	246	125	85

Table 4.3: Technical skill measurement by expert system

<b>Software Engineer</b>	<b>Total Points</b>	<b>Rank</b>
A	481	1
B	412	2
C	246	3
D	125	4
E	85	5

Table 4.4: Rank ordering of software engineers by expert system

### 4.3 Software Engineers Ranking by Manual Evaluation

Table 4.5 shows the technical skill points calculated by the expert system as well as by the manual evaluation reports. We also calculated the correlation coefficient between the points obtained by these two models. It has been observed that the Pearson correlation coefficient between the technical skill points generated by these two models is 0.953. Thus, the positive correlation coefficient value indicates strong positive relationship between the results obtained by expert system and manual evaluation.

It is evident from both the tables that software engineer ‘A’ holds the top rank regardless of the evaluation method used. Thus, we can draw the inference that the overall rank-ordering structure remains similar across both models.

Also, software engineer ‘E’ remains at the bottom position of the evaluations by both the models. Thus, it is safe to conclude that whatever model we use to establish the rank ordering of a software engineer based on their technical skills, the overall result remains the same for both models. The Appendix provides an elaborated study for the manual verification process.

Software Engineer	Total Points by Expert System	Total Points by Manual Evaluation
A	481	384
B	412	360
C	246	250
D	125	180
E	85	40

Table 4.5: Technical Skill assessment points by expert system and manual evaluation

## 4.4 Comparing Technical Skills of Software Engineers

Figure 4.1 presents the variation in the code commit signal of software engineers contributing to the 2.2.x version of spring boot project. It is evident that the total number of commits performed by ‘A’ every month is greater in comparison with the rest of the engineers.

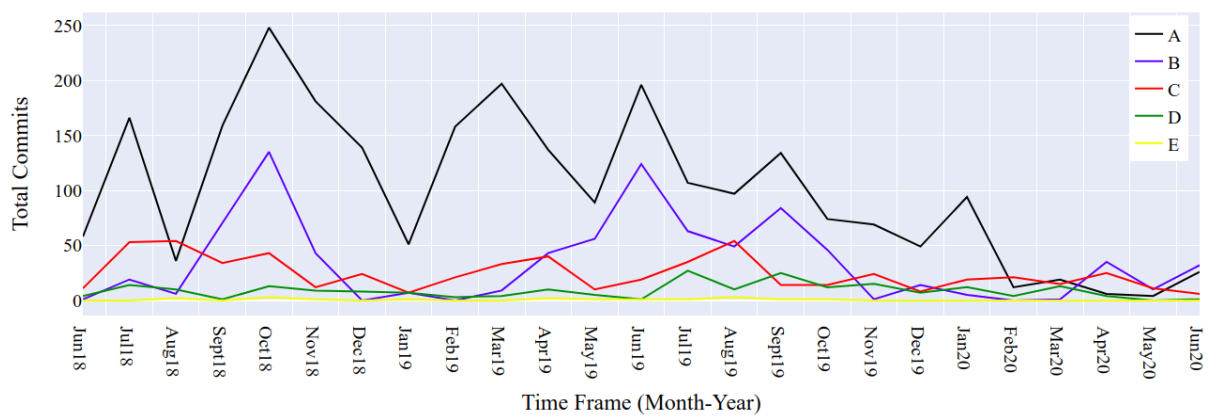


Figure 4.1: Comparing code commit signal of all software engineers (June 2018 - June 2020)

Figure 4.2 shows the variation in the code churn signal of software engineers contributing to the 2.2.x version of spring boot. As per the observed trend, code churn rate of both software engineers ‘A’ and ‘B’ reduces gradually.

Figure 4.3 shows the variation in the credits obtained for code stability signal by every software engineer, contributing to the 2.2.x version of the spring boot project. According to the figure, it is evident that code committed by software engineer ‘B’ in the month of October 2018 and August 2019 was the most stable and is still contributing to the project’s codebase as of June 19, 2020.

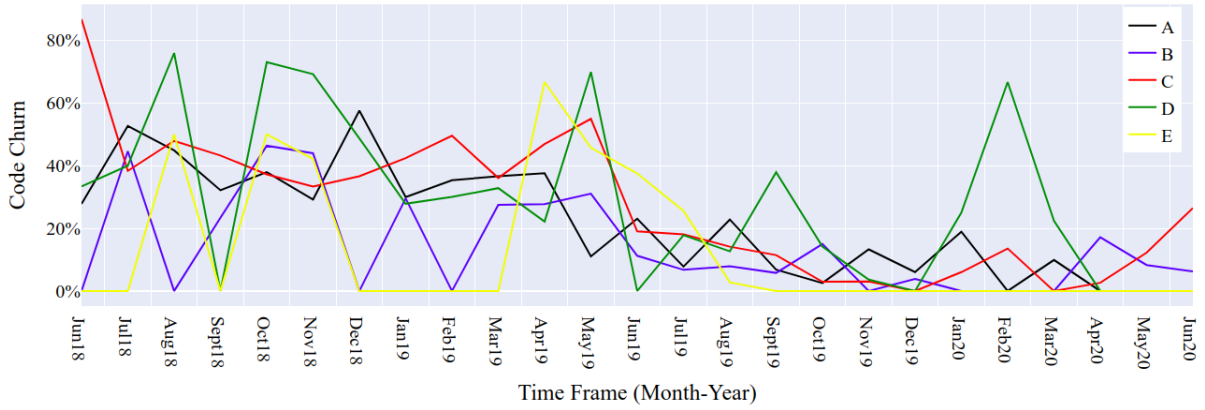


Figure 4.2: Comparing code churn signal of all software engineers (June 2018 - June 2020)

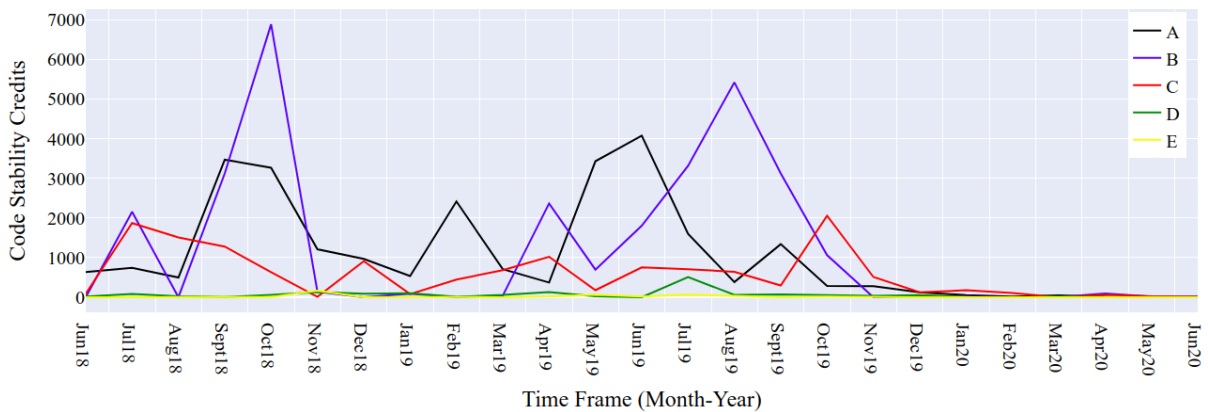


Figure 4.3: Comparing code stability signal of all software engineers (June 2018 - June 2020)

# 5 Conclusion

This chapter presents the conclusion of this research. It starts with highlighting that the goal of this research, as elaborated in chapter 1, has been accomplished. In this chapter, limitations of this study have also been discussed, followed by the mention of the contributions by this research. In the last part of this chapter, the future prospects for this thesis are also discussed.

## 5.1 Discussion

Traditionally, we were using resumes, LinkedIn profiles, and project team reports for assessing technical expertise of software engineers'. These self-authored resumes and profiles were not showing the real picture of one's skill level, sometimes hiding the horizon and sometimes overdoing it. Also, project team evaluation reports were also influenced by jealousy and opinions. Thus, this research was started with the primary goal of formulating a new comprehensive composite metric suite and moving forward from traditional practices.

The major challenge faced in this research was the evolving nature of software. One side an innovative software might go under daily code churn hammer and get moulded into a different one. In such a scenario, the code churn rate would be very high. On the other hand, a software that provides a stable solution should have a low code churn rate. Thus, we concluded that the selection of every metric is highly contextual.

In this research, we performed a series of experiments, one experiment for each qualitative signal representing the technical skills of a software engineer. Each successive experiment tried to refine the signal extracted in the previous experiment. We observed that no single metric was capable enough to assess the technical skills of a software engineer across all software development environments.

A Software engineer working on the innovation team exhibits qualitative signals which are applied in certain environments of work and a software engineer working on the legacy software systems such as banking, exhibits different standards where stability is

the most important. Thus, we concluded that signals are the same in both the working environments but their context varies as per the requirement.

Thus, we had to formulate an opinionated view to achieve the objective of creating an expert system capable of quantifying the technical skill of a software engineer. The opinionated view provided the context for every signal of the data model. Finally, with the help of a relative grading system, we rank ordered the technical skills of software engineers working in a software project.

## 5.2 Research Contribution

The section serves to throw light on numerous contributions by this research. The primary objective of this research was to create a data model, capable of measuring the technical skills of a software engineer. We already have qualitative metrics defined in the software engineering processes to measure the quality of a software system. Combining all those metrics to form a conceptualized metric suite might help us form the basis of a new era where we can quantify the qualitative aspects related to the technical skills of a software engineer. This research has contributed in many ways :

- Developed a new comprehensive data model to measure the technical skills of a software engineer as compared to traditional approaches that did lack on many layers.
- Discovered new and useful indicators from GitHub by analyzing the qualitative nature of software engineers' activities on the project's codebase that eventually helped to build the data model.
- Developed an expert system that rank orders the technical skills of a software engineer based on the opinionated view of the data model. The opinionated view represents a set of flexible heuristics applicable in different software development environments.
- Evaluated the efficiency of the data model by identifying the degree of correlation between the technical skill points obtained from the expert system and manual evaluation.
- Visualized a dashboard to compare different qualitative signals representing the technical skills of a software engineer within the team or a project.
- Demonstrated the applicability of the signals mined from OSS projects to quantify concepts such as skill assessment and productivity. It has been observed that the highly contextual nature of the extracted signals inhibits the creation of a generic data model that is valid across all software development environments.

## 5.3 Future Work

Measuring the technical skill of a software engineer is highly contextual, depending upon the evolving nature of the software. Some software might be up for innovation while some might be focusing on product stability, thus opening up numerous exploring possibilities. In the discussion section of every experiment, we observed the shortcomings of every qualitative signal. We also discussed different approaches to overcome these shortcomings and fine-tune the accuracy of the data model. For example, carefully defining the context for every signal, and then modifying the assumptions of the data model, improving the quality of the custom parser, and extending the library usage signal to quantify the importance of a software engineer within a team. Future work could also focus on understanding what other qualitative attributes can be quantified efficiently to assess the technical skill of a Software Engineer.

Since this research majorly focused on measuring the technical skills of a software engineer in Java technology, extending the support of this skill assessment framework to other technologies such as Python, Javascript, and Go would be an interesting initiative for future work.

Thus, many possibilities still need to be evaluated to develop a new cross-platform and cross-technology metrics that would be suitable to assess the technical skills of software engineers across various organizations and cultures.



# Bibliography

- [1] K. Juneja. Design of Programmer’s Skill Evaluation Metrics for Effective Team Selection. *Wireless Personal Communications*, May 2020. ISSN 1572-834X. doi: 10.1007/s11277-020-07517-6. URL <https://doi.org/10.1007/s11277-020-07517-6>.
- [2] Sample code commit signal on github ui. URL <https://github.com/rohan-tcd/test/commit/0ab2b0861b8cd746d889c3059855010deabfff6b>. Last Accessed: September 7, 2020.
- [3] G. A. Ferguson. On transfer and the abilities of man. *Canadian Journal of Psychology/Revue canadienne de psychologie*, 10(3):121–131, 1956.
- [4] T. H. Pear. The nature of skill. *Nature*, 122(3077):611–614, June .
- [5] J. P. Campbell, R. A. McCloy, S. H. Oppler, and C. E Sager. A theory of performance. *Personnel selection in organizations*, pages 35–70. San Francisco, CA: Jossey-Bass.
- [6] S. Hawk, K. M. Kaiser, T. Goles, C. V. Bullen, J. C. Simon, C. M. Beath, and K. Frampton. The information technology workforce: A comparison of critical skills of clients and service providers. *Information Systems Management*, 29(1):2–12, 2012.
- [7] M. Jørgensen. Failure factors of small software projects at a global outsourcing marketplace. *Journal of Systems and Software*, 92:157–169, June 2014.
- [8] Linkedin corporation: Social networking service, . URL <https://www.linkedin.com/>. Last Accessed: September 7, 2020.
- [9] N. Fenton and B. Kitchenham. Validating software measures. *Journal of Software Testing, Verification, and Reliability*, 1(2):27–42, 1991.
- [10] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, 1996.
- [11] T. DeMarco. Controlling software projects: Management, measurement and estimation. 1982.

- [12] Github: Git repository hosting service, . URL <https://github.com/>. Last Accessed: September 7, 2020.
- [13] Tejaswini, S. Patil, S. Salimath, V. Naik, R. Nandi, M. S. Patil, I. Bidari, and S. Chickerur. Programmer productivity analyzer tool. In *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICIC)*, pages 1–8, 2017.
- [14] P. L. Li, A. J. Ko, and J. Zhu. What makes a great software engineer? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 700–710, 2015.
- [15] S. Lee, D. Hooshyar, H. Ji, K. Nam, and H. Lim. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing*, 21(1):1097–1107, March 2018. ISSN 1573-7543. doi: 10.1007/s10586-017-0746-2. URL <https://doi.org/10.1007/s10586-017-0746-2>.
- [16] T. S. Somasundaram, U. Kiruthika, M. Gowsalya, A. Hemalatha, and A. Philips. Determination of competency of programmers by classification and ranking using ahp. In *2015 IEEE International Conference on Electro/Information Technology (EIT)*, pages 194–200, 2015.
- [17] X. Li, P. Shih, and E. David. The effect of software programmers’ personality on programming performance. In *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 209–213, 2018.
- [18] R. N. Reinstedt. Results of a programmer performance prediction study. *IEEE Transactions on Engineering Management*, EM-14(4):183–187, 1967.
- [19] Fitria and I. G. B. B. Nugraha. Formation of software programmer team based on skill interdependency. In *2018 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 77–81, 2018.
- [20] F. Ortin, O. Rodriguez-Prieto, N. Pascual, and M. Garcia. Heterogeneous tree structure classification to label java programmers according to their expertise level. *Future Generation Computer Systems*, 105:380 – 394, 2020. ISSN 0167-739X. doi: <https://doi.org/10.1016/j.future.2019.12.016>.
- [21] European e-competence framework.
- [22] The european qualifications framework for lifelong learning (eqf). URL [http://relaunch.ecompetences.eu/wp-content/uploads/2013/11/EQF\\_broch\\_2008\\_en.pdf](http://relaunch.ecompetences.eu/wp-content/uploads/2013/11/EQF_broch_2008_en.pdf). Last Accessed: September 7, 2020.

- [23] G. R. Bergersen, D. I. K. Sjøberg, and T. Dybå. Construction and validation of an instrument for measuring programming skill. *IEEE Transactions on Software Engineering*, 40(12):1163–1184, 2014.
- [24] G. Rasch. Probabilistic models for some intelligence and achievement tests. 1960.
- [25] R. D. Luce and J. W. Tukey. Simultaneous conjoint measurement: A new type of fundamental measurement. *Journal of Mathematical Psychology*, 1(1):1–27, 1964.
- [26] T. G. Bond and C. M. Fox. Applying the rasch model: Fundamental measurement in the human sciences. 2001.
- [27] D. Wilking, D. Schilli, and S. Kowalewski. Measuring the human factor with the rasch model. *Balancing Agility Formalism in Software Engineering*, 5082:157–168, 2008.
- [28] P. Pirolli and M. Wilson. A theory of the measurement of knowledge content access and learning. *Psychol*, 105(1):58–82, 1998.
- [29] A. Syang and N. B. Dale. Computerized adaptive testing in computer science: Assessing student programming abilities. 25(1):53–56, 1993.
- [30] T. Honglei, S. Wei, and Z. Yanan. The research on software metrics and software complexity metrics. In *2009 International Forum on Computer Science-Technology and Applications*, volume 1, pages 131–136, 2009.
- [31] N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., USA, 3rd edition, 2014. ISBN 1439838224.
- [32] H. R. Bhatti. *Automatic Measurement of Source Code Complexity*. PhD thesis, 2010.
- [33] D. I. De Silva, N. Kodagoda, and H. Perera. Applicability of three complexity metrics. In *International Conference on Advances in ICT for Emerging Regions (ICTer2012)*, pages 82–88, 2012.
- [34] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.
- [35] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697–708, 1987.
- [36] Upwork. URL <https://www.upwork.com/>. Last Accessed: September 7, 2020.
- [37] Hackerrank, . URL <https://www.hackerrank.com/>. Last Accessed: September 7, 2020.

- [38] Coderwalle. URL <https://coderwall.com/>. Last Accessed: September 7, 2020.
- [39] Standardizing technical hiring with hackerrank’s new skills-based experience, . URL <https://blog.hackerrank.com/standardizing-technical-hiring-with-hackerrank-skills/>. Last Accessed: September 7, 2020.
- [40] Waydev: Developer summary. URL <https://waydev.co/features/developer-summary/>. Last Accessed: September 7, 2020.
- [41] Misra c. URL [https://en.wikipedia.org/wiki/MISRA\\_C](https://en.wikipedia.org/wiki/MISRA_C). Last Accessed: September 7, 2020.
- [42] Sonarqube: Code quality and security. URL <https://www.sonarqube.org/>. Last Accessed: September 7, 2020.
- [43] lint (software) - wikipedia, . URL [https://en.wikipedia.org/wiki/Lint\\_\(software\)](https://en.wikipedia.org/wiki/Lint_(software)). Last Accessed: September 7, 2020.
- [44] Thank you for 100 million repositories, . URL <https://github.blog/2018-11-08-100m-repos/#:~:text=Today%20we%20reached%20a%20major,collaborating%20across%201.1%20billion%20contributions>. Last Accessed: September 7, 2020.
- [45] Linux kernel source tree, . URL <https://github.com/torvalds/linux>. Last Accessed: September 7, 2020.
- [46] Spring boot. URL <https://github.com/spring-projects/spring-boot>. Last Accessed: September 7, 2020.
- [47] An open source machine learning framework for everyone. URL <https://github.com/tensorflow/tensorflow>. Last Accessed: September 7, 2020.
- [48] Github is your new resume, . URL <https://code.dblock.org/2011/07/14/github-is-your-new-resume.html>. Last Accessed: September 7, 2020.
- [49] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona. Effort estimation by characterizing developer activity. *Proceedings of the 2006 international workshop on Economics driven software engineering research*, (1):3–6, 2006.
- [50] E. Kalliamvakou, K. Blincoe, L. Singer, D. M. German, and D. Damian. An in-depth study of the promises and perils of mining GitHub. 21:2035–2071, October 2016. ISSN 1573-7616. doi: 10.1007/s10664-015-9393-5. URL <https://doi.org/10.1007/s10664-015-9393-5>.

- [51] F. Chatziasimidis and I. Stamelos. Data collection and analysis of github repositories and users. In *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 1–6, 2015.
- [52] Github developer: Rest api v3, . URL <https://developer.github.com/v3/>. Last Accessed: September 7, 2020.
- [53] Top 11 applications of java with real-world examples. URL <https://techvidvan.com/tutorials/applications-of-java/>. Last Accessed: September 7, 2020.
- [54] The top programming languages 2019. URL <https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019>. Last Accessed: September 7, 2020.
- [55] Spring boot: Apache license 2.0. URL <https://github.com/spring-projects/spring-boot/blob/master/LICENSE.txt>. Last Accessed: September 7, 2020.
- [56] Spring boot 2.2 release notes. URL <https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.2-Release-Notes>. Last Accessed: September 7, 2020.
- [57] G. Gousios, E. Kalliamvakou, and D. Spinellis. Measuring developer contribution from software repository data. In *Proceedings of the 2008 international working conference on Mining software repositories, MSR '08*, pages 129–132, Leipzig, Germany, May 2008. Association for Computing Machinery. ISBN 9781605580241. doi: 10.1145/1370750.1370781. URL <https://doi.org/10.1145/1370750.1370781>.
- [58] Y. Qiu, W. Zhang, W. Zou, J. Liu, and Q. Liu. An empirical study of developer quality. In *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*, pages 202–209, 2015.
- [59] Y. Wu, Y. Yang, Y. Zhao, H. Lu, Y. Zhou, and B. Xu. The influence of developer quality on software fault-proneness prediction. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 11–19, 2014.
- [60] Github glossary. URL <https://docs.github.com/en/github/getting-started-with-github/github-glossary#commit>. Last Accessed: September 7, 2020.
- [61] The repo commits api: List commits, . URL <https://developer.github.com/v3/repos/commits/#list-commits>. Last Accessed: September 7, 2020.
- [62] B. M. y. Parareda. Measuring productivity using the infamous lines of code metric. 2007.

- [63] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co, Redwood City, CA, United States, 1986. ISBN 978-0-8053-2162-3.
- [64] S. Mukherjee. A survey on metrics, models tools of software cost estimation. *International Journal of Advanced Research in Computer Engineering Technology*, 2: 2620–2625, 09 2013.
- [65] The repo commits api: Get a commit, . URL <https://developer.github.com/v3/repos/commits/#get-a-commit>. Last Accessed: September 7, 2020.
- [66] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, USA, 2 edition, 2008. ISBN 0321356683.
- [67] Git blame. URL <https://git-scm.com/docs/git-blame>. Last Accessed: September 7, 2020.
- [68] Pmd source code analyzer project. URL [https://pmd.github.io/pmd/pmd\\_userdocs\\_extending\\_writing\\_rules\\_intro.html](https://pmd.github.io/pmd/pmd_userdocs_extending_writing_rules_intro.html). Last Accessed: September 7, 2020.
- [69] Khoshgoftaar and Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proceedings 1994 International Conference on Software Maintenance*, pages 58–67, 1994.
- [70] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.
- [71] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005.
- [72] M. C. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin. Code decay analysis of legacy software through successive releases. In *1999 IEEE Aerospace Conference. Proceedings (Cat. No.99TH8403)*, volume 5, pages 69–81 vol.5, 1999.
- [73] J. Shimagaki, Y. Kamei, S. McIntosh, D. Pursehouse, and N. Ubayashi. Why are commits being reverted?: A comparative study of industrial and open source projects. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–311, 2016.
- [74] Git log. URL <https://git-scm.com/docs/git-log>. Last Accessed: September 7, 2020.

- [75] T. Menzies, S. Williams, O. El-Rawas, B. Boehm, and J. Hihn. How to avoid drastic software process change (using stochastic stability). In *2009 IEEE 31st International Conference on Software Engineering*, pages 540–550, 2009.
- [76] W. B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.
- [77] D. Threm, L. Yu, S. Ramaswamy, and S. D. Sudarsan. Using normalized compression distance to measure the evolutionary stability of software systems. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 112–120, 2015.
- [78] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, 1989.
- [79] E. Reshidi. *Is Relative Grading an Incentive Towards Segregation?* PhD thesis, 2014.
- [80] Dash open source: Dash enterprise. URL <https://plotly.com>. Last Accessed: September 7, 2020.
- [81] Plotly python open source graphing library. URL <https://plotly.com/python/>. Last Accessed: September 7, 2020.

# A1 Code Commit and Code Change Signal

## A1.1 What is Commit?

Commit is a save point that records changes to the source code files with additional audit information such as branch name, author name, commit time, and commit message to document the changes for better understanding of the commit. Figure A1.1 shows a commit information through GitHub's user interface.

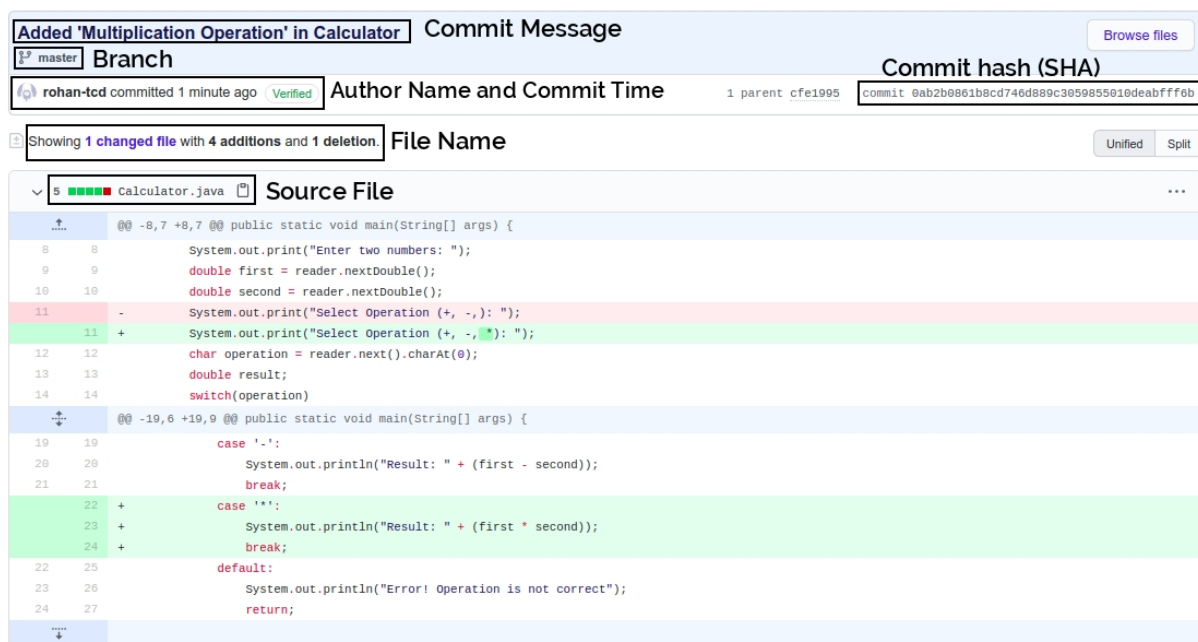


Figure A1.1: Code Commit Signal on GitHub UI [2]

## A1.2 What is Code Change?

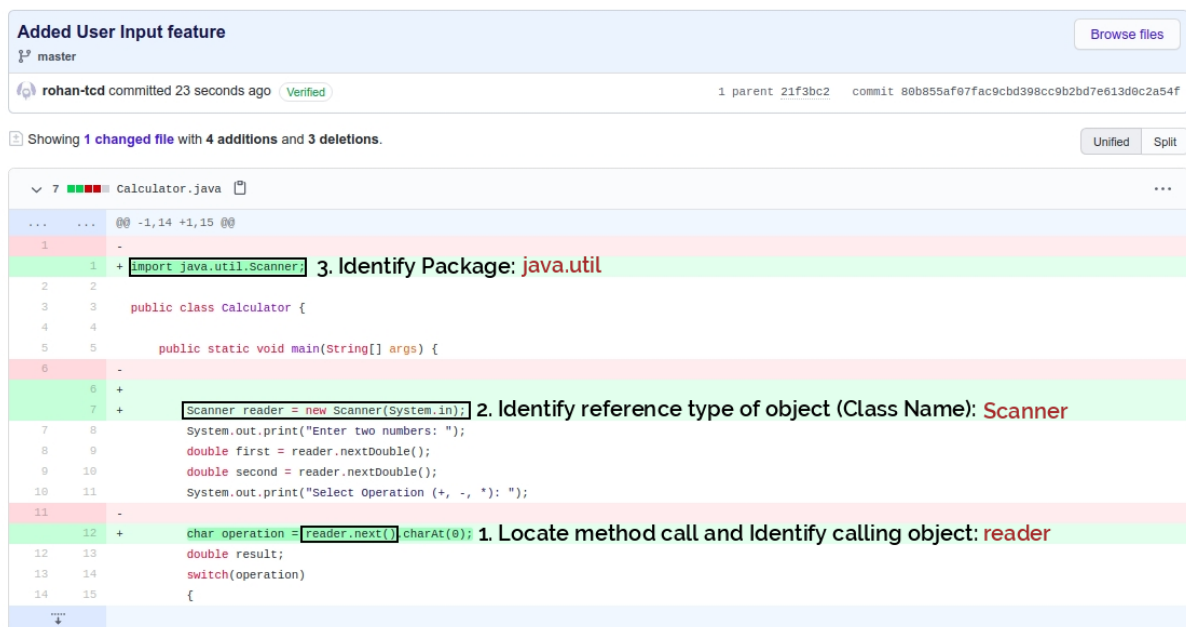
Code Change represents the part of the source code that have been modified in a commit. Figure A1.1, indicates that there are total of 5 lines of code change (4 addition and 1



deletion). **GitHub represents modification to a line of code as a combined activity of deletion and addition.** Green highlighted lines indicate the newly added lines of code and red highlighted lines indicate the deleted lines of code.

## A2 Measuring Library Usage Signal

As discussed in the section 3.2.3 , we programmatically tried to measure the library (both inbuilt and external) usage of a software engineer in Java technology.



The screenshot shows a GitHub commit titled "Added User Input feature" by user rohan-tcd. The commit message is "Showing 1 changed file with 4 additions and 3 deletions." The file being changed is Calculator.java. The code is shown with line numbers and changes indicated by '+' and '-' signs. Three specific lines are highlighted in green, each with an annotation:

- Line 1: `import java.util.Scanner;` with annotation "3. Identify Package: java.util"
- Line 7: `Scanner reader = new Scanner(System.in);` with annotation "2. Identify reference type of object (Class Name): Scanner"
- Line 12: `char operation = reader.next().charAt(0);` with annotation "1. Locate method call and Identify calling object: reader"

Figure A2.1: Extracting Library Usage Signal [2]

Figure A2.1 indicates the steps involved in extracting the library usage signal.

1. Identify the modified lines of code in a given commit (indicated by green highlighted lines in figure A2.1).
2. Locate the method call and identify the calling object.
3. Identify the reference (Class name in case of Java technology) of the calling object.
4. Identify the Package name or Library to which it belongs.

# A3 Measuring Code Churn Signal

As discussed in the section 3.2.4, we programmatically identified the number of lines of code that have been refactored (or deleted) as of June 19, 2020, since its initial commit date.

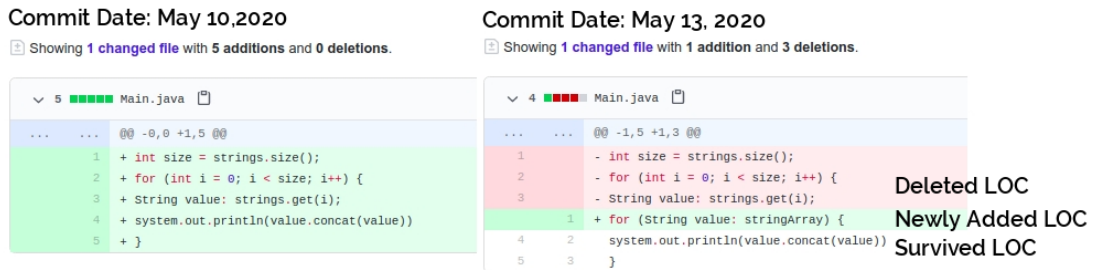


Figure A3.1: Code Churn Example [2]

Figure A3.1 indicates two commits performed by an engineer on the same source code file. Initial commit, dated May 10, 2020, contains 5 lines of code which are added newly to the codebase. On May 13, 2020, engineer optimized the same code by refactoring the code committed previously. Thus from the commit performed on May 10, 2020, there are only 2 lines of code which are productive and still contributing to the project.

Thus, we can calculate the code churn rate for commit performed on May 10, 2020 as:

$$CodeChurnRate := \frac{NumberOfLOCDeleted}{NumberOfLOCAdded} * 100 \quad (1)$$

$$CodeChurnRate := \frac{3}{5} * 100 = 60\% \quad (2)$$

From equation 2, we can conclude that commit performed on May 10, 2020 has **60% code churn rate**.

## A4 Measuring Code Stability Signal

As discussed in the section 3.2.5, code stability signal accurately weighed the importance of every stable line of code in a commit. In this experiment, we calculated the code stability signal using formula 1:

For example, in figure A3.1, we identified that there are only 2 lines of code which are survived till May 13, 2020 since its initial commit on May 10, 2020. Thus, the code stability credit points for commit performed on May 10, 2020 can be calculated as:

$$\textit{CreditPointsForCodeStability} = \textit{NumberOfLOCsSurvived} * \frac{\textit{NumberOfDaysElapsed}}{365} \quad (1)$$

$$\textit{CreditPointsForCodeStability} = 2 * \frac{3}{365} = \mathbf{0.01} \quad (2)$$

Similarly, if we assume that these 2 lines did not get modified till January 1, 2021 then these 2 survived lines would receive

$$\textit{CreditPointsforCodeStability} = 2 * \frac{236}{365} = \mathbf{1.29} \quad (3)$$

credits for code stability.

Although application of code stability signal is highly contextual, this signal significantly measures the technical skills of a software engineer.

# A5 Measuring Technical Skills of Developer ‘A’ using Manual Evaluation Framework

N.B. Technical skills measured through manual assessments are subjected to the imitated knowledge of the human expert (in this research, the author himself). The appendix discusses the thought process and logical reasoning of the human expert while quantifying the qualitative signals through manual assessment.

According to the table 4.2, software engineer ‘A’ has the highest count of commits within the period of 2 years from June 18, 2018, to June 19, 2020. Since this signal was extracted directly using GitHub APIs, the manual analysis was performed to understand the value delivered to the project by these commits. Thus, commits are analyzed to understand if they are performed to resolve a pull request, merge request, or an issue.

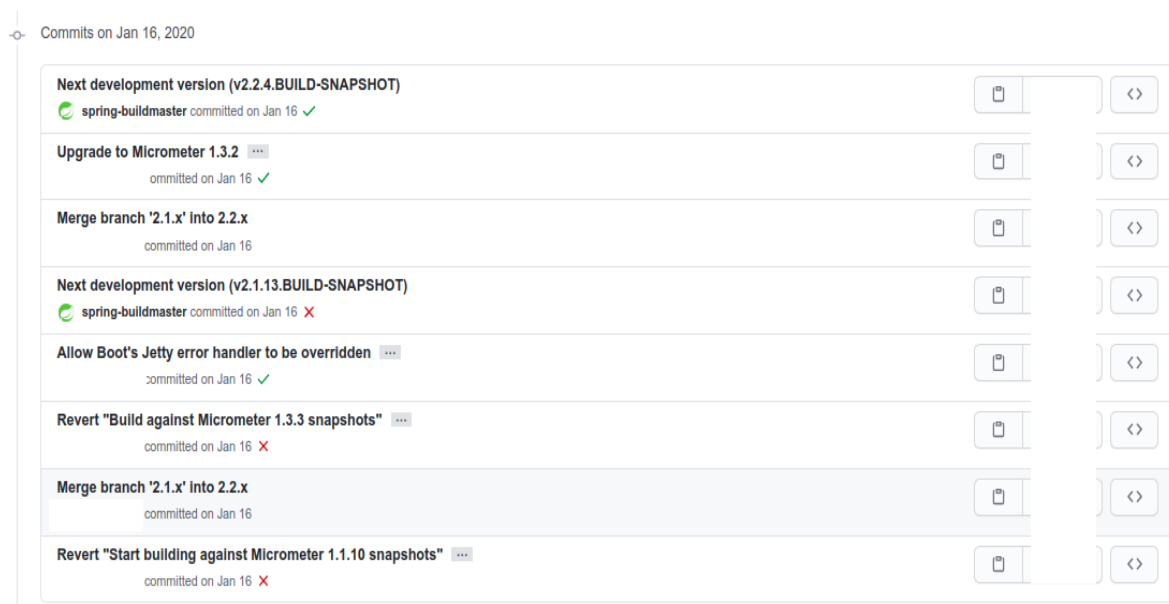


Figure A5.1: Different variety of commits performed by ‘A’

It was observed that the developer 'A' has majorly contributed by resolving bugs in 2.1.x version of the Spring Boot project and then merging those changes in the version 2.2.x. Also, many of these commits were related to updating the external dependencies used in the Spring Boot project.

To analyze the Code Change signal, a manual assessment of some of the commits was performed. It has been observed that developer 'A' has effectively contributed by not only writing the efficient java code but also documenting the changes for better maintainability of the project.

```

... .. @@ -1,5 +1,5 @@
1 1 /*
2 2 - * Copyright 2012-2017 the original author or authors.
3 3 + * Copyright 2012-2018 the original author or authors.
4 4 *
5 5 * Licensed under the Apache License, Version 2.0 (the "License");
6 6 * you may not use this file except in compliance with the License.
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 17
18 18
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29
30 30
31 31
32 32 * configuration.
33 33 * <p>
34 34 * The configuration will not be activated if {@literal spring.aop.auto=false}. The
35 35 - * {@literal proxyTargetClass} attribute will be {@literal false}, by default, but can be
36 36 - * overridden by specifying {@literal spring.aop.proxyTargetClass=true}.
37 37 + * {@literal proxyTargetClass} attribute will be {@literal true}, by default, but can be
38 38 + * overridden by specifying {@literal spring.aop.proxy-target-class=false}.
39 39 *
40 40 * @author Dave Syer
41 41 * @author Josh Long
42 42
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
51 51
52 52
53 53
54 54
55 55
56 56
57 57
58 58
59 59
60 60
61 61
62 62
63 63
64 64
65 65
66 66
67 67
68 68
69 69
70 70
71 71
72 72
73 73
74 74
75 75
76 76
77 77
78 78
79 79
80 80
81 81
82 82
83 83
84 84
85 85
86 86
87 87
88 88
89 89
90 90
91 91
92 92
93 93
94 94
95 95
96 96
97 97
98 98
99 99
100 100

```

Figure A5.2: Documentation performed by 'A'

In the case of library usage signal, it has been clearly observed that the custom parsers could not extract complex method calls from the code changes. For example, method invocations using 'this' variable in Java, invocations in method chaining, and invocations using the latest features of Java 8 such as Stream API, Functional Interfaces, Lambda, method referencing using double colon operator '::'. Thus, the accuracy of the library usage signal in the automated expert system needs to be improved.

The code churn and the code stability signal were analyzed manually to understand the reliability of every changed line of code. It has been observed that most of the commits performed by 'A' tried to resolve the existing issues. There were almost negligible occurrences of reopening the issue resolved by 'A'. Table A5.1 represents the technical skills of developer 'A' measured through manual evaluations.

```

339 +     private final ResourceProperties resourceProperties;
340 +
341 +     FaviconConfiguration(ResourceProperties resourceProperties) {
342 +         this.resourceProperties = resourceProperties;
343 +     }
344 +
Method invocation using 'this' keyword in Java

339 345     @Override
340 346     public void addResourceHandlers(ResourceHandlerRegistry registry) {
341 347         if (!registry.hasMappingForPattern("favicon.ico")) {
342 -             registry.addResourceHandler("favicon.ico").addResourceLocations("classpath:favicon.ico");
343 +             registry.addResourceHandler("favicon.ico")
344 +                 .addResourceLocations(this.resourceProperties.getStaticLocations())
345 +                 .addResourceLocations("classpath:favicon.ico");
Method Chaining
343 351     }

```

Figure A5.3: Inefficiency of custom parsers to extract complex method calls: Method Chaining

```

@@ -380,8 +380,11 @@ void customContentNegotiatingViewResolver() {
380
381     @Test
382     void faviconMapping() {
383 -         this.contextRunner
384 -             .run((context) -> assertThat(getResourceMappingLocations(context).get("/favicon.ico")).hasSize(1));
385 +         this.contextRunner.run((context) -> {
386 +             List<Resource> favIconResources = getResourceMappingLocations(context).get("/favicon.ico");
387 +             assertThat(favIconResources.stream().map(ClassPathResource.class::cast).map(ClassPathResource::getPath))
388 +                 .containsExactly("META-INF/resources/", "resources/", "static/", "public/", "favicon.ico");
389 +         });
Method referencing using '::' operator

```

Figure A5.4: Inefficiency of custom parsers to extract complex method calls: Method Referencing using '::' operator

Software Engineer	A
Code Commit Signal	90
Code Change Signal	74
Library Usage Signal	60
Code Churn Signal	80
Code Stability Signal	80
<b>Total Points</b>	<b>384</b>

Table A5.1: Technical skill points of developer ‘A’ measured through manual evaluations