# Measuring Software Engineering Report

Tomasz Bogun 19335070

## Introduction

The measurement of software engineering is an interesting task. Like most things, humans have a desire to quantify and measure everything to understand it better. Sometimes this task is straight forward but sometimes it is not. Software engineering is probably one of the most difficult professional disciplines to measure people's performance in. However, it is possible if it is done carefully. There are many metrics we can measure about a software engineer, but any one metric cannot be the solely used to measure the work of a software engineer because it does not show the full picture. There are several reasons why we would like to measure software engineering. Employers could use it to identify the best and worst performing employees in their team and help them make decisions based on this. Software engineers themselves could use it to find their strength and weaknesses. Customers could use it to analyse a team of software engineers before committing to hire them.

## 1.Measurable Data

### 1.1. Lines of Code

The number of lines of code a software engineer contributes is the simplest metric we can observe and would probably be the first metric to come into mind to a non-software engineer. This is a big problem because many customers, and some managers involved in the software engineering process might not understand the discipline completely and might be misled by thinking this metric accurately shows a software engineers performance. This is the opposite from the truth. In software engineering, it is much preferred to write a clear and concise piece of code rather than a very long, and therefore more complicated piece of code that does the same thing. If this metric was solely used to measure the performance of software engineers, it would lead to less efficient code that would not be able to understanded by other team members than the clear concise counterpart. It would not only annoy software engineers if knew that their performance is being measured based on the lines of code they contribute, rather than the quality of it. But also, it would allow them to manipulate the metric to receive a more favourable performance ranking. Software engineers would feel more inclined to write more bloated, and hence less understandable and less efficient code to increase this metric. Generally, it is better to contribute a piece of code that has less lines than another piece of code that does the exact same thing. However, it would also not be wise to measure the performance of software engineers based on how little lines of code they write. This is because it is possible to shorten down a piece of code to extremely small lengths by methods such as decreasing the length of variable names, cramming more code into one line than separating it out clearly across a few lines etc. This would degrade the clarity of the written code and would make it much more difficult for other team members to understand and use. Because of this, there needs to be a fine balance between the number of lines of code a software engineer contributes and the clarity of it. It would be better to write code that is slightly longer but much more understandable by team members. Because of these things, this metric generally should be rarely used, and if it is used, it should be used with extreme care and understanding of software engineering.

## 1.1. Comments and Documentation

The process of software engineering is not just writing code in front of a computer all day. Most software engineering is done in teams. And for this to happen, excellent communication is required. Software engineering is a very cognitive task that needs to be simplified as much as possible so that it can be understood by other people quicker. This can be code by writing clear, short code with plenty of comments. Comments can be thought of as on-the-go documentation for software engineers that are using the code of other people. They are little text descriptions beside lines of code that describe the purpose of the code beside it. This is useful for when the code is a little complex and is not evident from the very start what it exactly does. A software engineer might also put instructions about how the code works and how it should be used by other people, along with specific things to be aware of. This greatly increases code clarity and allows other software engineers to use code that was made by other people, quicker. Rather than going through every line and deducing logically what a piece of code does, a software engineer would much rather look and understand it from clearly written comments beside it in simple language. Comments greatly improve teamwork efficiency and reduce colleagues' confusion. This metric is quite valuable because software engineering is mostly a teamwork profession. Generally, more comments are better, but they should be clear and understandable and should not confuse other people even more.

## 1.3. Efficiency

Software engineering is not just the process of writing code and making sure it works. A software engineer should strive to make their code as efficient as possible. This is because efficient code will run faster and take up less space and will be able to be run on more systems. This effect will be compounded even more if a person's code will be called from other people's code multiple times. Any software engineer could write a piece of code that does what it is meant to, but it takes a good software to write that code as efficiently as possible. This is because they need to understand the inner workings of their code completely, not just what appears on the surface. A good software engineer might write code that is much more efficient than another piece of code that does the same thing, and that might not appear that much more efficient to other people just by looking at it. It is often the case that in a software engineering team environment, only results matter, and customers often don't even think about efficiency if the software works as they expected. Because of this, some software engineers tend to neglect the efficiency of the code they write. But people often don't realise that this degrades the application more than they expect. Just because a piece of software works and is responsive now, it doesn't mean that it will be so responsive in the future when more features are added, and the inefficient code is built upon. Therefore, it is necessary to write software that is as efficient as possible from the start. Some software development methodologies prefer to leave the efficiency work until the end of the process, but it often takes too much work and redesigning to make the old code more efficient if it is already deeply embedded in the software. Altering it could possibly mean altering any other pieces of code that use the inefficient code. The most common way to assess the piece of code would be the popular big O notation that measures how well the code scales.

## 1.4. Bugs and Errors

One of the most important things about software engineering is that it simply works as expected. The software could be written clearly, with good comments, documentation, and efficiency, but all this this doesn't matter if the software doesn't work or has even minor bugs. A customer often only cares about the user experience of the software and doesn't worry about the inner workings if the software works well. The best way to prevent bugs is to implement unit testing from the very beginning of the process. This will ensure that the code written will work in all circumstances. Software testing by simply using the software and looking at the result is not enough. When we use a piece of software, it is highly unlikely that every line of code will be run and be therefore tested. Unit testing aims to test every single line of a certain piece of code and every possibility that can arise from running it. If this is done correctly, it will significantly reduce the number of errors and bugs in the final software.

### 1.5. Activity

The activity of a software engineer could also be measured in many ways. Generally, a more active programmer is more productive as they are consistently working on a project. The activity of a software engineer could be measured in some cases by the frequency of the commits that they contribute. If a software engineer commits frequently, then it most likely means that they are quite active. But the opposite is not necessarily true. If a software engineer commits less often, it does not mean that they are not active, it might mean that they are working on a big or difficult task that takes a longer amount of time, which will reduce the frequency of their commits. The activity of a software engineer could also be measured in different ways such as how fast they reply to questions, how fast they review pull requests or the amount of time they are active on their computer.

### 1.6. Impact

The impact of a piece of code aims to measure how much work was contributed to a software engineering project. This goes far beyond measuring only lines of code. When calculating code impact, we need to consider other properties about the contribution such as the severity of changes, the number of files that have been modified/added/deleted, the number of lines deleted, the location of these modifications, how this change affects other areas of the software. Impact aims to answer how much cognitive effort did the software engineer put in in a certain contribution. Generally, a higher impact score means a more productive and contributing engineer.

## 2. Platforms

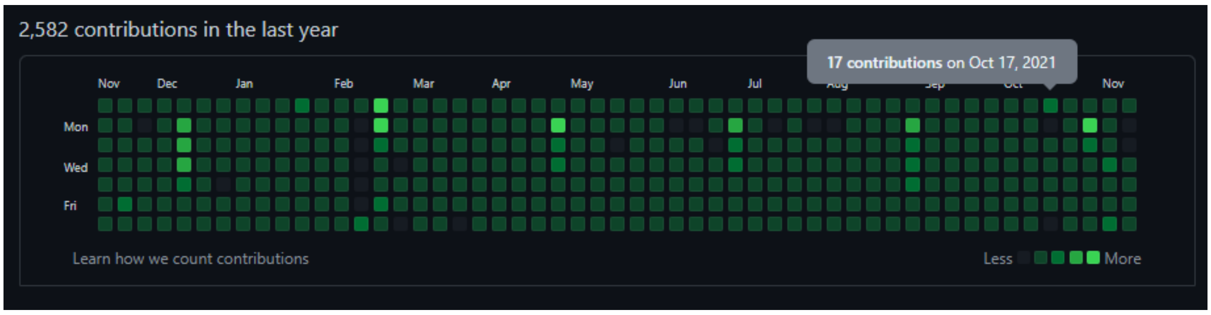
There are multiple platforms that we can use to gather and process metrics about software engineering. There has been a big increase over the years in the number of platforms that are available to gather measurable data about software engineering. These platforms can process this data and extract several metrics that might be of interest. Some of these platforms are free but some of them can be quite expensive.

### 1.1. Github

The most popular version control system based on git is by far Github. It is so popular in the world of software engineering that nearly all software engineers will have at least heard of it. If a software engineer ever used a version control system, it is most likely Github because it has an 83.12% market share in the source code management market. This is because it is easy to use and packed with features. Conveniently, it also has some data-gathering, processing and visualisation functionality built right into it on its website. Github is often overlooked for gathering data about software engineering metrics, so it is quite surprising that free software like Github can provide us with so many useful fundamental metrics.

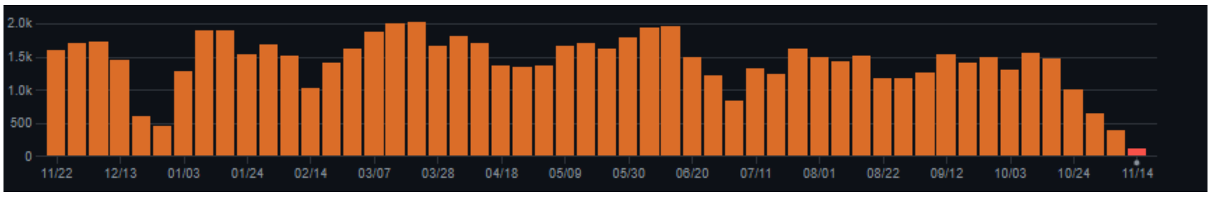
### 1.1.1. User Activity

If we would like to see the historical activity of a user on Github, we can look at their profile page and we will be able to see how active they are very quickly. This activity is measured in terms of commits a day. A grey box means that there were no commits that day, and a dark green box means that there were more commits than a day that has a light green box. This information should be interpreted cautiously as it shows only the number of commits, and a big commit will be treated the same as a much smaller commit. However, we could deduce if a person is always consistently working on new and existing projects, or if they only work on projects occasionally, by looking at the gaps between commits. For example, an employed software engineer that works in a company that uses Github as their version control system will most likely has a fuller and greener commit history than a hobbyist programmer that creates projects in his free time. The picture below shows the activity of Linus Torvalds on Github as of 20/11/2021:

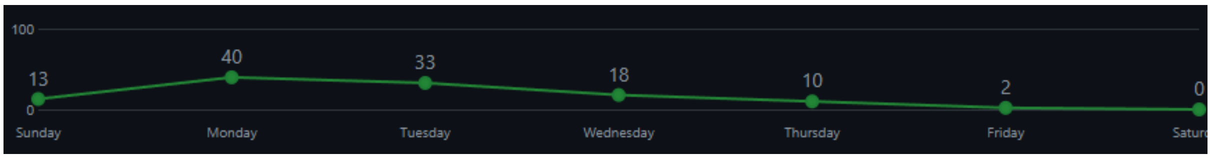

2,582 contributions in the last year
17 contributions on Oct 17, 2021
Nov
Dec
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Mon
Wed
Fri
Learn how we count contributions
Less
More

### 1.1.2. Repository Activity

We can also see the historical activity of an individual repository on Github. The activity of a repository is also counted in the number of commits. This can be easily viewed in the repository's insights>commits section. The picture below shows a visualisation of this data for the Linux Github repository as of 20/11/2021:

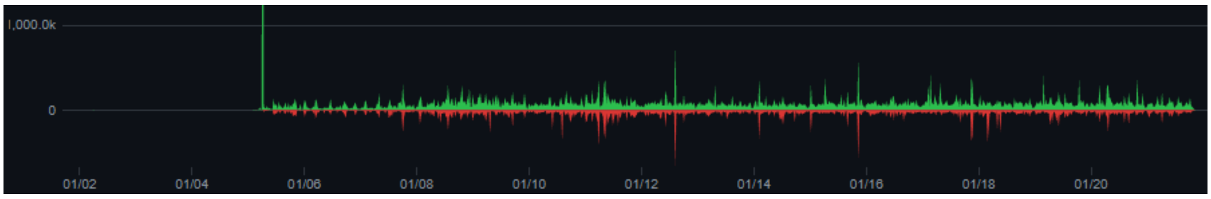


We can also see what days in the past week were the most active. As expected, the most active days are the weekdays, and the least active days are the weekends. This data can be quite useful for companies that have weekly software engineering goals. If there tends to be more commits at the start of the week than the end of the week, then it is an indication that the software engineering team tends to manage their time well and get their task done swiftly. On the contrary, if most of the commits happen on the last working day, then it might be an indication that the team ran into time trouble and rushed to get the task done before the deadline. This data can also be viewed from the repository's insights>commits section. This The picture below shows a visualization of this data on the Linux Github repository for the week of 14/11/2021:



### 1.1.3. Lines of Code – Additions and Deletions

The data amount of code additions and deletions can also be easily gathered and visualised straight from Github itself. As mentioned earlier in the report, this data should be interpreted carefully because more code additions do not mean more productivity on a certain day. However, this data can be useful in some cases. For example, if there are much more deletions than additions on a certain day, it might mean that a large feature was deleted from the project. It could also mean that existing code has been redesigned to use less lines of code than before to improve on its readability or efficiency. On the contrary, if there are much more additions than deletions on a certain day, it might mean that the software engineering team has just begun working on a new feature in the project. This data can be accessed from the repository's insights>Code frequency section. The picture below shows the historical number of code additions and deletions for the Linux Github repository as of 20/11/2021 (The first commit reaches to over 6,000k but that is cropped from the picture):



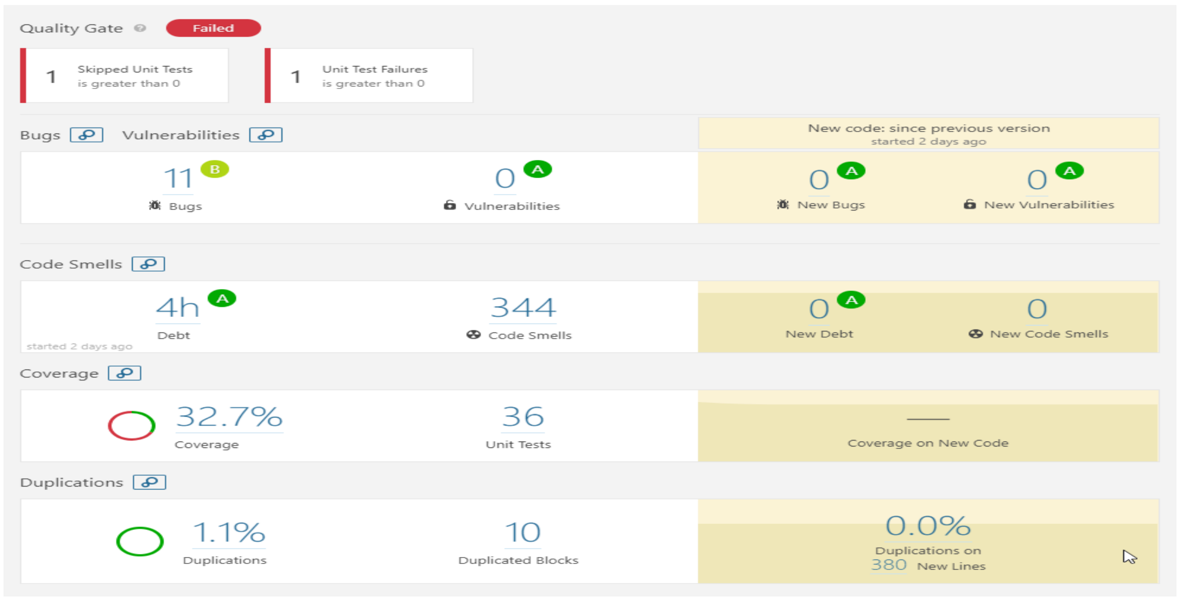
## 2.2. SonarQube

SonarQube is an open-source platform for automatically and continuously analysing the quality of code. It supports over 20 programming languages. It can generate reports on several metrics such as code duplication, unit testing, code complexity, code efficiency and comments. However, it is most famous and used for detection of bugs and vulnerabilities in software.

The detection of bugs in software before release is extremely important, especially in software that will be used by a large amount of people. It would be a commercial disaster to discover that a piece of software is full of unexpected bugs and does not always work as expected, after it is released. This could severely damage the reputation of the company that has developed the software, which could decrease the size of the user base or the number of future software customers.

It would be even worse to discover that a piece of software has security vulnerabilities that could be exploited by malicious users to gain access to functionality or data that they should not have access to. Security vulnerabilities like this have the potential to disrupt the software service, leak sensitive information among other things. It doesn't take much negligence to introduce security vulnerabilities, and even security professionals can introduce security vulnerabilities by accident from times to time. If it was so easy to prevent this, then there would not be so many vulnerabilities in software nowadays. SonarQube aims to help prevent common security vulnerabilities by analysing submitted code. This has an enormous potential to prevent devastating damages by simple mistakes such as not parsing user input correctly, misconfiguration of network connectivity, missing encryption, missing authorisation etc.

SonarQube provides a feature called Quality Gate that is a set of preconfigured conditions that test a software's readiness for release. It analyses multiple metrics about the code and gives either a passing or failing grade, with a breakdown of each of the metrics. Below is a picture of the Quality Gate dashboard report:


Quality Gate Failed

1 Skipped Unit Tests is greater than 0

1 Unit Test Failures is greater than 0

Bugs Vulnerabilities

New code: since previous version started 2 days ago

11 B Bugs

0 A Vulnerabilities

0 A New Bugs

0 A New Vulnerabilities

Code Smells

4h A Debt

started 2 days ago

344 Code Smells

0 A New Debt

0 New Code Smells

Coverage

32.7% Coverage

36 Unit Tests

— Coverage on New Code

Duplications

1.1% Duplications

10 Duplicated Blocks

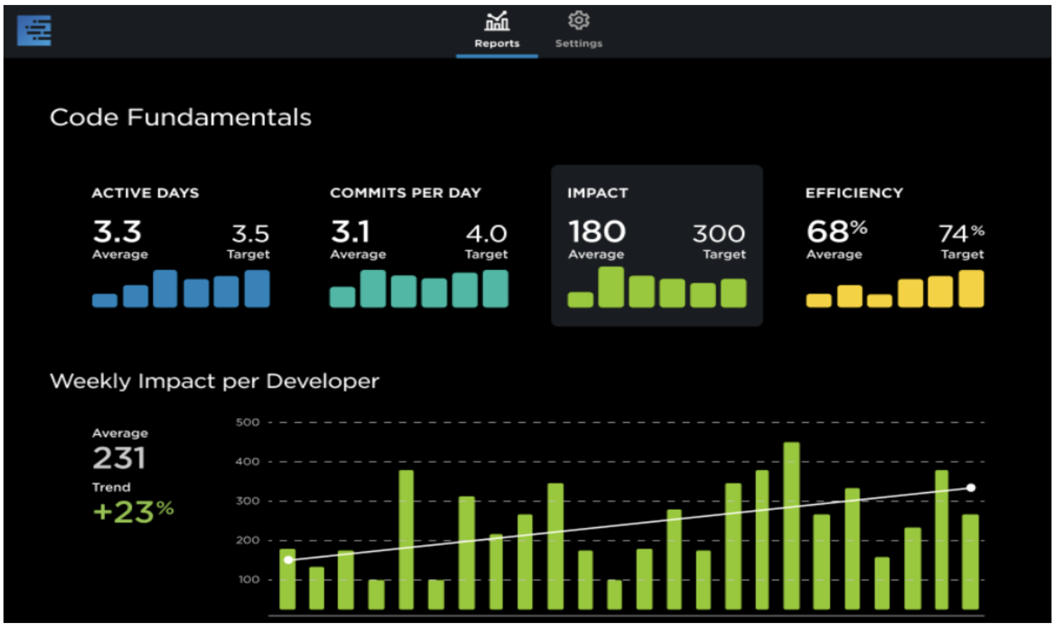
0.0% Duplications on 380 New Lines

## 2.3. Pluralsight flow

Pluralsight flow is another great platform for automatically analysing software. It was previously called GitPrime but has rebranded since then. It can provide us with many useful metrics about a software engineering project. Pluralsight gathers data from a git version control system like Github and presents it visually in a dashboard report. This is a very popular platform among large teams of software engineers. This is because it performs complex calculations to extract useful metrics that are not so easy to extract and analyse accurately from a git version control and presents it very clearly and intuitively in such a way that almost anyone could understand, even if they are not so familiar with software engineering themselves. This platform would be especially useful for customers that are purchasing a piece of software and would like to be informed of the progress of the work, without having to look at overly technical metrics. This is a premium platform with quite expensive pricing, starting at $499 per active contributor per year at the time of writing, but there are even more expensive options. It is often worth the large investment in this platform for large software engineering teams with plenty of resources but not so much for smaller teams.

Plural sight flow can collect a range of metrics, but it is most known for analysing code's impact and efficiency. Efficiency is extremely important in software engineering, especially at the start of the process. Because of this, it is useful to see at a quick glance what efficiency score a project achieves, without having to analyse the code and deduce it manually. We can also set targets for each of these metrics and periodically check if the software engineering team meets these targets. Pluralsight flow can also extrapolate trends on each of these given metrics, so we could see if any of the provided metrics are decreasing or increasing over time. This can be useful to see if changes such as new policy, goals, training, or employees in our software engineering team has an impact on any of these metrics over time.

Below is a picture of an example visualisation dashboard on the Pluralsight flow platform:

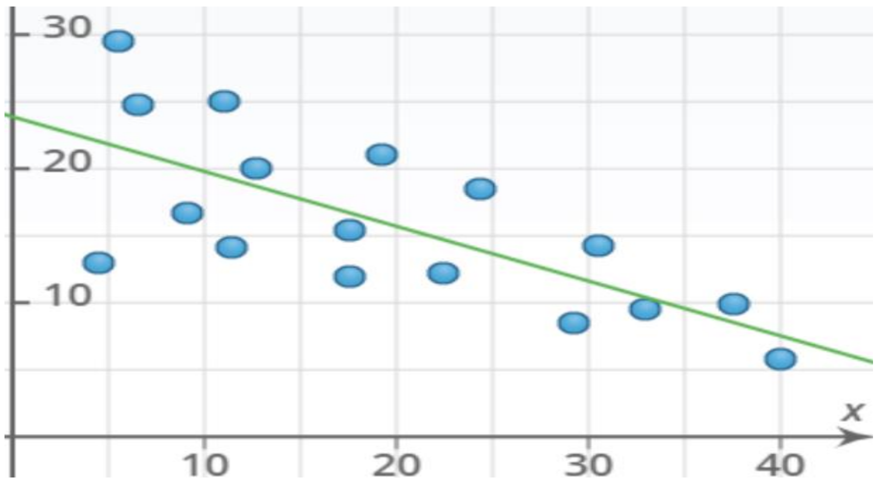


## 3. Algorithmic Approaches

There are a vast number of ways to use the data and metrics we can gather about software engineering. We can process these metrics using various algorithms to extract data that might be of interest. We could use this data to make more informed decisions by trying and identify patterns and trends from these metrics.

### 3.1. Correlation

Trying to identify how one thing affects the other is called correlation. Identifying correlation has many different uses. In software engineering we use the concept of correlation to see how one metric is linked to another metric. This could be particularly useful if we have several different metrics about software engineering, but we are trying to gather another certain metric that is difficult to compute or gather. Instead, we could sometimes use correlation to attempt to predict this metric based on other metric data we have. For example, we might want to see if there is a link between the years of experience of a software engineer and the number of bugs or errors they introduce. One would expect that a more experienced software engineer will produce less bugs and errors in their code. To check this hypothesis, we could gather the number of years of experience, and the number of bugs introduced by every software engineer in a large data set and plot it on a scatter plot. The x-axis would represent on metrics and the y-axis would represent the other metric. Then, we could look at the resulting graph and try identifying a pattern or a trend between these two metrics. It is often the case that there is a clear pattern visible straight away, which means that the metrics are related in some way. In other cases, we could algorithmically calculate the line of best fit of these data points. Correlation is a useful tool for measuring software engineers and can be used to attempt to find a link between any two metrics. This could be also extended to three metrics by plotting the data points in three dimensions. For example, if a recruiter is looking for a software engineer that introduces the least number of bugs among anything else, they might want to perform a correlation analysis like the one described above. They might discover that experienced software engineers are much less likely to introduce bugs than less experienced software engineers. Based on this information, the recruiter might be willing to pay more for an experienced software engineer. The picture below is a hypothetical example of a scatter plot of software developing experience (x-axis) vs bug frequency (y-axis). In this example, there is a clear pattern that more experiences software engineers contribute less errors and bugs from the collected data.



This line of best fit can be easily computer algorithmically, by repeatedly trying different values for the slope and origin of the line, and seeing which line minimises the average distance between the points and the line.

## 3.2. Machine learning

There has been a huge increase in the widespread use of machine learning for data processing. This is because machine learning requires demanding computation, which is now more accessible. Machine learning is a part of AI (Artificial Intelligence). Instead of having fixed algorithms, machine learning aims to learn and improve from previous data examples. It can identify complex patterns and trends in a dataset that a human could never notice. For machine learning to work best, it requires a large dataset to “train” on. There are three different types of machine learning, supervised machine learning, unsupervised machine learning, and reinforcement learning. Supervised machine learning is most used in data analytics. It can be separated into different categories of regression and classification. It is called supervised learning because it is supervised by an existing data set that it trains on and tries to create a model that most accurately captures how the inputs of the data are related to the outputs.

Regression aims to reveal the relationship between different variables. It works by accepting several inputs and tries to predict an output based on the inputs. The goal is to predict the output of new inputs that are not in the training dataset as accurately as possible. It is widely used for making projections. It would be possible to create a machine learning program that takes several different software engineering metrics that were discussed previously as inputs and tries to predict another metric like efficiency of a software engineer, or a team of software engineers based on this training data.

Classification aims to assign given data into multiple categories that have something in common. Similarly, to regression, classification attempts to draw a conclusion from given inputs, based on previously supplied training data. Unlike regression machine learning that can produce a range of different outputs, classification has a limited number of outputs because it classifies data into a limited number of categories or classes. An example of classification is spam filtering in emails, using the text of the email as an input, a classification machine learning program has only two options, to classify it as spam or not. Similarly, we could create a machine learning program that could try predicting whether a software engineer will miss a certain deadline or not, based on previous data.

Machine learning is incredibly useful for identifying trends among large pieces of data. It is so good at this that it is sometimes able to identify complex patterns and relationships between different input variables that humans would never have been able to discover because of the complexity. For example, we might discover that the reliability of a software engineer increases as they gain experience, but not if they work in groups of 2-5 people, unless the team has an average of over 5 years of experience in software engineering. This is an arbitrary example that a machine learning program would be able to identify with sufficient data, but it would be difficult for a human to discover this complex relationship.

As we can see, machine learning can be useful for assisting with measuring some metrics of software engineering. However, there is one certain thing we need to consider when implementing machine learning in measuring software engineering. Machine learning tends to generalise data. It works by analysing large data sets and tries to identify patterns from the data. But each piece of data in a data set it unique and sometimes machine learning fails to see important details and context about the data. That is, software engineers are humans and humans are quite difficult to predict. Every software engineer has different experiences, culture, attitudes, outlooks, backgrounds. Because of this, there is no such thing as an "average software engineer", as every aspect of a person would have to be exactly average which is almost impossible. It would be unfair to try predicting a metric of an individual software engineer using machine learning that makes conclusions from large data sets that other people generated. It would be even more unfair to make employment decisions using the results from machine learning. Because of this, machine learning should be used with great caution and understanding when measuring software engineering.

## 4. Ethical Concerns

When we try to measure software engineering, there arise several ethical concerns, especially if we measure the software engineering of other people without their knowledge.

### 4.1. Reliability

Software engineering is one of the most difficult professions to accurately measure a person's performance in. There are many metrics to consider when measuring a software engineer. However, these metrics must be used with extreme care and understanding of the software engineering process when evaluating the performance of a software engineer, as any one of these metrics only show a measure relating to one aspect of the process, it doesn't show the full picture. This is an ethical concern because employers may be using incorrect metrics to evaluate their software engineering team and might be making decisions about their employment even if they don't have a complete understanding of these different metrics and how they should be used. It would be highly unethical to judge a software engineers performance based on a single metric without looking more at the details, especially if the quirks of the individual metric are not fully understood and accounted for. A software engineer that scores better in one metric than another software engineer but might score lower in another metric. Because of this, it might sometimes be necessary to look at each person's performance on an individual basis rather than looking at metrics generated by software.

### 4.2. Privacy

Privacy is an ethical concern in measuring software engineering. Some people would be uncomfortable with aspects of their work being collected, stored, and processed. They might be worried that this data is being shared with other companies for evaluation or being compared to data generated by other software engineers. They most likely not know what data is stored, where it is stored, how it is processed and used, and who has access to it. There is a rise in the number of companies that want their employees to wear monitoring devices that track their heart rate, movement, location. This is supposedly meant to improve the experience of an employee by trying to detect their stress levels, reminding them to take a break etc. However, most people would probably be very uncomfortable with this extremely personal data being collected about them for workplace reasons.

## 4.3. Measurement in the workplace

Employers usually need to evaluate the performance of their employees, to make informed decisions. They would like to know which employees performs well and which don't. Therefore, the measurement of software engineering can be a tool for measuring the performance of employees in this area. When software engineering is being measured in the workplace, it raises ethical concerns almost every way it is done. There are three main possibilities for the measurement of performance of software engineers in the workplace

### 4.3.1. Employees unaware of the measurements

If employees are unaware that their software engineering is being measured using various metrics, they will perform normally as they usually would, and the measurements will be quite accurate. However, there are obvious ethical questions arising from this. Is it fair to measure an employee without them knowing? Most people would probably not be happy to see that their work was being measured without them knowing what is exactly being measured about them. They would most likely prefer to know about this measurement in advance so that they can prepare for it. They might also worry about either the accuracy, reliability, privacy, or processing of these metrics.

### 4.3.2. Employees aware of the exact measurement

Employees would much prefer to know whether their work is being measured or not. They would also like to know what exactly will be measured about their work before it is measured. This would make them less stressed and worried about their privacy if they knew exactly what is being measured. However, this might lead to manipulated results. If software engineers know exactly what will be measured about them, they will be likely to write code in such a way to manipulate the metric in their favour. This might cause their work to drop quality in other metrics that they know are not measured that might be important in the software engineering process. This doesn't necessarily raise any serious ethical concerns, but the accuracy and validity of the measurement is subject to being manipulated favourably by the employees, which defeats the purpose of the measurement to an extent.

### 4.3.3. Employees aware of the measurements but unaware of the details

Employees that are informed that their software engineering is being measured but are not told the exact metrics and details will always have this in the back of their mind. They might be stressed about their work being measured less favourably based on things that they don't even know are measured, and therefore can't control. This would create an atmosphere of uncertainty especially if the metrics are used to make decisions about employment. This uncertainty might even decrease the productivity of the engineers and the quality of the software they produce. This is an obvious ethical concern as it could make the employees constantly worry about this unknown measurement about them. It might also cause them to overwork themselves and spend too much time perfecting their work in every aspect so that it will be measured favourably for as many metrics as possible.