

CSU33012 Software Engineering

Measuring Engineering Report

**Stephen Davis, Std# 18324401**

**November 14, 2021**

Contents

[1 Introduction 1](#_Toc88839902)

[2 How Can Software Engineering be Measured? 2](#_Toc88839903)

[2.1 Lines of Code 2](#_Toc88839904)

[2.2 Number/Frequency of Commits 2](#_Toc88839905)

[2.3 Leadtime 3](#_Toc88839906)

[3 What Platforms Can Be Used to Gather and Process Data? 3](#_Toc88839907)

[3.1 Pluralisation (Git Prime) 3](#_Toc88839908)

[3.2 WayDev 4](#_Toc88839909)

[3.3 Code Climate 5](#_Toc88839910)

[**3.3.1** **Velocity** 5](#_Toc88839911)

[**3.3.2** **Quality** 5](#_Toc88839912)

[4 What Algorithms Can We Use? 6](#_Toc88839913)

[4.1 Halstead Complexity Measures 6](#_Toc88839914)

[4.2 Artificial Intelligence 7](#_Toc88839915)

[4.3 Computational Intelligence 7](#_Toc88839916)

[4.4 Cyclomatic Complexity 7](#_Toc88839917)

[5 Is This Ethical? 8](#_Toc88839918)

[5.1 Privacy 9](#_Toc88839919)

[5.2 Increasing Productivity 9](#_Toc88839920)

[6 Conclusion 9](#_Toc88839921)

# Introduction

This report will discuss why one would want to measure software engineering, how software engineering can be measured, what platforms can be used to measure software engineering and what algorithms we can use to carry out such measuring. This report will then discuss the ethics of measuring software engineering, before finally concluding.

# How Can Software Engineering be Measured?

There is much debate about whether software engineering can actually be measured. A popular view, and my personal view, is not whether software engineering can be measured, but how can software engineering be measured? It is worth noting that there are many ways we can measure software engineering, but not every way is accurate, insightful and fair. Most of the time the issue with these inaccurate measurements is that they lack context. It is worth noting that each measurement comes with its own advantages and disadvantages. Some examples of how to measure software engineering productivity which I will now discuss are:

* Lines of Code
* Number/Frequency of Commits
* Number of Code Reviews Done
* Leadtime

## Lines of Code

The most obvious method to measure software engineering is to measure the number of lines of code a software developer writes. This metric is considered to be “The Original Code Metric” (<https://www.gitclear.com/blog/the_4_worst_software_metrics_agitating_developers_in_2019> ). The advantages for this method are that it is very simple to measure and easy to understand this measurement. You do not need a technical background to understand how to analyse this metric.

Although there are some advantages to using Lines of Code (LOC) as a measurement, they are far out-weighed by the disadvantages. The most important disadvantage of LOC, is that it is “a very easy metric to game” (<https://www.pluralsight.com/blog/teams/lines-of-code-is-a-worthless-metric--except-when-it-isn-t-> ). Figure 1 and 2 below illustrates how one could gamify their code to appear more productive.

Figure (for loop)

Figure (same as above by copying and paste same line of code)

As can be seen by the above examples, LOC as a metric promotes inefficient, sloppy code and is far too easy to game. We must remember, “You don't pay a Michelangelo to make brush strokes, you pay him to be a genius” (<https://www.pluralsight.com/blog/teams/lines-of-code-is-a-worthless-metric--except-when-it-isn-t-> ).

## Number/Frequency of Commits

At first, measuring the number of commits made by an engineer may appear to be beneficial as a metric for measuring productivity, however, just like LOC, it is easily gamified. Our hope would be that software engineers commit code when they have made progress on a project, and by comparing the number of commits made by different developers, the higher the number, the more productive an engineer is.

However, in theory, two engineers could essentially write the same code, except the first engineer commits the code when he has the functionality working, and let’s say the second engineer commits every time he adds a new line of code. Now, of course, we do not want the latter to happen, but what is stopping this engineer, or any engineer, from doing so? Again, this illustrates how gamification can easily be utilised to improve your score when being analysed by the number of commits you make.

The disadvantages of using the number of commits made by an engineer is that it encourages insignificant and unnecessary commits. “You are incentivising them to make a commit every time they author a line of code” (<https://www.gitclear.com/blog/the_4_worst_software_metrics_agitating_developers_in_2019> ). The size and value of the commit is not taken into account, and this is why it is so easy to gamify. Perhaps one of the cruellest disadvantages of using the number/frequency of commits as a metric is that “if you're a hard-working developer that's striving to solve as many issues as possible, by simply saving their work more often, your lazier coworker will shoot past you on the commit count leaderboard?” (<https://www.gitclear.com/blog/the_4_worst_software_metrics_agitating_developers_in_2019> ). This metric rewards those who can game the system and those who play fairly appear to be the workers lacking in productivity. Simply put, “Commit count creates a toxic atmosphere among developers”.

One advantage of using this metric is that if you notice it has been a while since an engineer has made a commit, this is “often a signal that they may be stuck” (<https://www.gitclear.com/blog/the_4_worst_software_metrics_agitating_developers_in_2019> ). However, it is quite clear that this one advantage is negligible when compared against all the disadvantages associated with this metric.

Again, as software engineers, we favour quality of code over quantity.

## Leadtime

“Lead time quantifies how long it takes for ideas to be developed and delivered as software” (<https://stackify.com/track-software-metrics/> ). To measure Leadtime, you “need to have a clear definition of when work begins and ends” (<https://humanitec.com/blog/lead-time-a-key-metric-in-devops> ). The idea behind using Leadtime as a metric is to measure how quickly engineers or a team of engineers can go from idea to production. If we notice one project idea has a significantly longer Leadtime than another, we may ask why? Perhaps a project idea was underspecified or there were permission requests which took a long time to come back. The advantages of this metric are that it incentivises software engineers to “improve how responsive they are to customers” (<https://stackify.com/track-software-metrics/> ). Another advantage is that a company may realise from analysing this metric, that their Leadtime for releasing a fully-functional working version of the idea is too long, and their competitors may have beat them to the punch. By analysing the Leadtime, team leads may decide to “release little and often; you release each feature as it is ready, rather than waiting for an “all-in” big release” (<https://humanitec.com/blog/lead-time-a-key-metric-in-devops> ). This adapted approach would then keep the customers engaged and loyal to your company’s product.

With any measurement, if you measure the wrong things, you risk pushing the quality of code down, which tends to reward mediocrity. Therefore, it is vital that if you are measuring software engineering, you choose a fair, insightful and well-thought-out metric. Do not just measure software engineering for the sake of measuring.

# What Platforms Can Be Used to Gather and Process Data?

Nowadays we have version control systems such as GitHub, which collect vast amounts of data. There is no doubt that this data exists, and that we have access to this data, but the question is how can we gather and process the specific data that we want. Thankfully, there are various platforms which provide such a service. The examples I will discuss in this report are:

* Pluralisation (Git Prime)
* WayDev
* Code Climate
* Personal Software Process

## Pluralisation (Git Prime)

As previously mentioned, we can measure software engineering by the lines of code (LOC), the number of commits, Leadtime and various other metrics. These metrics individually are not much use and certainly not very insightful in the quest of measuring software engineering. However, by combining various different metrics, we can draw meaningful conclusions about our productivity as developers.

Pluralisation flow, previously called Git Prime, is a platform designed to collect various sources of information such as lines of code, pull requests, etc. from Github’s wealth of data on software engineers, and analyse these metrics collectively, to produce meaningful insights.

Flow collects and configures historical git data into easy-to-understand insights and reports to assist software engineering teams in becoming more successful. Flow empowers you to be able to identify bottlenecks so you can remove them, compare trends and help your team reach their full potential. Based on the Github data flow collects and analyses from your team, it can then recommend skills which your team may need to develop

“If we were not using Flow, it would be like going back to the Stone Age. We've gotten used to the metrics and the visibility Flow provides us” (Rob Teegarden, VPE, Dealersocket).

Pluralisation believes Git is more than version control. It is a record of how your team works. To align with this value, flow provides a powerful visualization into your teamwork dynamics. Questions such as what percentage of pull requests or commits get zero responses, what percentage of the team is involved in feedback and are senior engineers providing feedback and mentorship, are all answered by flow’s powerful visualisation features.

A precarious position to find your team in is when your most informed, and up to speed developer(s), the central link in the team, is unavailable. Suddenly your team members may find themselves stuck, where no present members have the answer.



Figure 1: If the central link in your team becomes unavailable, the whole chain breaks down. This is where knowledge sharing reports come in handy.

Knowledge Sharing reports in Flow help you measure and improve how codebase knowledge is distributed across your team and empowers you to distribute value evenly across your engineering team.

(<https://www.pluralsight.com/product/flow> )

## WayDev

The next platform on the list is WayDev. If you are looking to become more productive and happier with your work, then WayDev should be on the top of your list of platforms available to gather and process data. Earlier on in this report, we saw that a hard-working engineer would appear to be less productive than a lazy one, who knew how to game the system. However, with WayDev, we have justice. WayDev pride themselves on the fact that you can rest assured your achievements will be acknowledged. By being justly rewarded for all your hard work, you will be motivated to work harder, since you know your work is appreciated. This is a win-win situation, the productivity of the software engineer improves which is a win for the company, and the deserved engineer receives credit, acknowledgement and a sense of pride for his/her impactful efforts.

Another feature WayDev provides is for you to ship your code faster, by equipping you with the tools necessary to visualise the codebase, pull requests, and tickets your team produce from sprint to sprint. WayDev allows you to analyse how you spend your valuable time, how often you send code for review, and identifies those suggestive areas of improvement.

Like flow, WayDev unlocks the true value of data available to us on GitHub, and empowers us to learn, grow and improve, both individually and collectively, from this data.

<https://waydev.co/software-engineers/>

## Code Climate

Code Climate advertises that you can “leverage [their] data-driven insights to build a culture of trust and high performance” (<https://codeclimate.com/> ).

Code Climate understand the importance of meaningfully analysing data and put emphasis on how one can “help every member of [their] team excel”.

There are two main products offered by Code Climate:

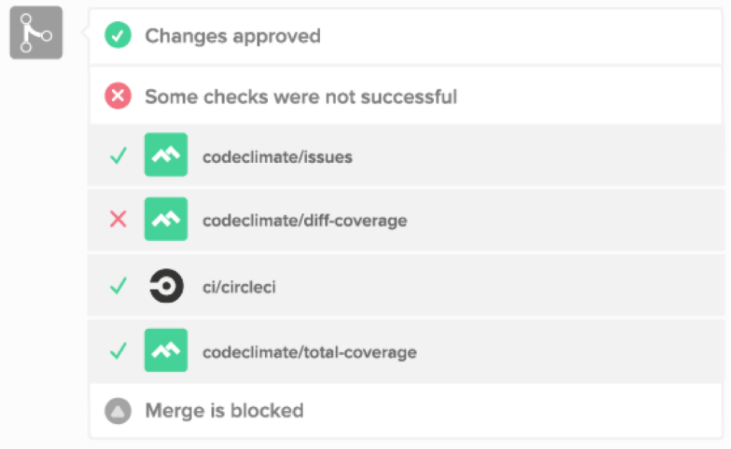
1. Velocity
2. Quality

### **Velocity**

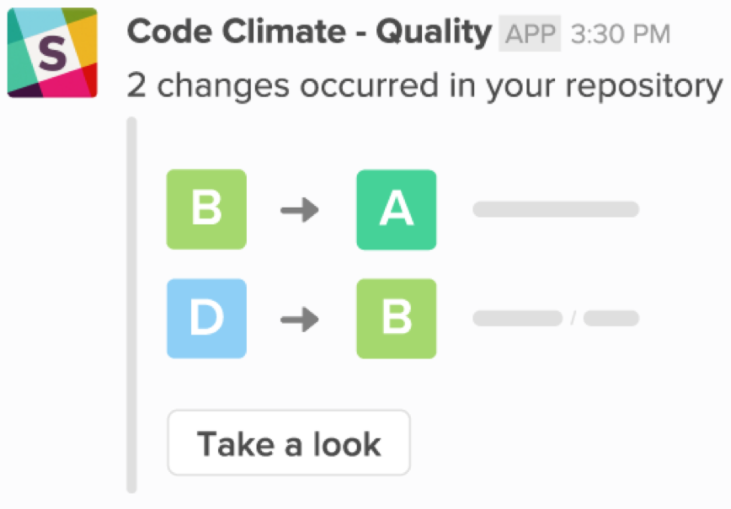
Velocity is a powerful product offered by Code Climate, which has many overlapping features provided by Pluralsight and WayDev, such as insights into how you spend your time, recommendation on what skills your team can improve on etc. Since these features have already been discussed previously, I will move on to Quality, the second product offered by Code Climate.

### **Quality**

Quality is a product offered by Code Climate which focuses on test coverage. The way this product works is it grades code out of ten by running a suite of designed tests. The key benefit of these tests is that Code Climate will block a merge if tests fail, preventing problematic merges.



As mentioned in the beginning of this report, when being analysed under lines of code, software engineers can simply game the system. With Quality, this cheat-code is effectively dealt with, since the product neatly displays what files have changed, how much they have changed and how exactly they have changed.



If the same files appear to be changing again and again, perhaps this is an indication that a developer is in need of assistance. The power of Quality is that it saves an incredible amount of time that would have otherwise been spent manually reviewing code. Also, humans are susceptible to error, whereas machines running a suite of tests will not have room for human error. Quality prevents gamers from short-cutting there way to the top of the leader board, and ensures good programming practice and honesty is adhered to.

<https://codeclimate.com/>

# What Algorithms Can We Use?

Once we have collected all the data, we need to measure software engineering, we then need to find some way to interpret this data in a meaningful way. Thankfully, there are numerous algorithms available in the software realm. The algorithms which I will discuss in this report are:

* Halstead Complexity Measures
* Cyclomatic Complexity
* Maintainability Index

## Halstead Complexity Measures

Maurice Howard Halstead introduced the algorithmic Halstead Complexity Measures in 1977 (<https://en.wikipedia.org/wiki/Halstead_complexity_measures> ). Halstead believed a computer program is considered to be a collection of tokens, which may be classified as either operators or operands, and that all software science metrics can be defined in terms of these basic symbols (<https://www.javatpoint.com/software-engineering-halsteads-software-metrics> ).

To begin using this algorithm, it is necessary to look at the basic measures which will be used in subsequent calculations. These are:

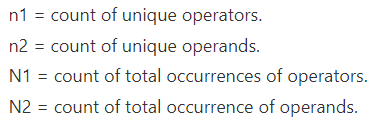


Figure 2: The Halstead complexity measures.

After establishing what the Halstead Complexity measures are, one can then progress to look at the Halstead metrics listed below.

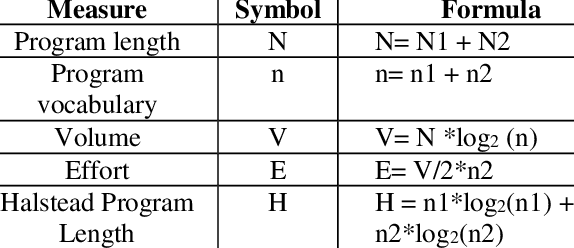


Figure 3: The Halstead Complexity Metrics, alongside their corresponding symbols and formulae. Source: https://www.researchgate.net/figure/Halstead-Complexity-Metrics\_tbl1\_319481865

(<https://www.javatpoint.com/software-engineering-halsteads-software-metrics> )

## Cyclomatic Complexity

Developed by Thomas J. McCabe, Sr, in 1976, “cyclomatic complexity is a software metric used to indicate the complexity of a program” (<https://en.wikipedia.org/wiki/Cyclomatic_complexity> ). What cyclomatic complexity of a section of source code measures is the number of linearly independent paths within this section of code.

The formula for calculating the cyclomatic complexity of a section of code, which we represent with the letter M, is:

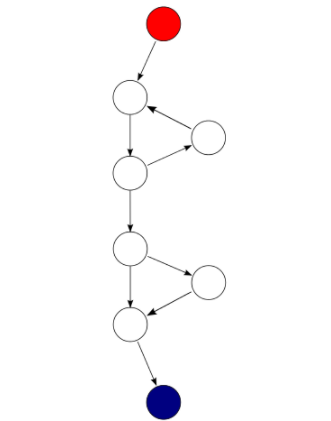
M = E − N + 2P,

where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.



The logic behind cyclomatic complexity is that the more decisions that have to be made in code, the more complex it is.

At the beginning stages of this report, lines of code was discussed as a metric, and pointed out to be easily gamed. LOC was linked to bad programming practice, where developers simply write more lines of code, and in the software engineering world, a popular opinion is that the more lines of code in a function, the more likely it is to have errors. However, when you combine cyclomatic complexity with lines of code, then you have a much clearer picture of the potential for errors. Just because you have more lines of code, it does not necessarily mean you are more prone to having errors in your code.

As an algorithmic approach, cyclomatic complexity can be used to “get a sense of how hard any given code may be to test, maintain, or troubleshoot as well as an indication of how likely the code will be to produce errors” (<https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022> ). The value in cyclomatic complexity as an algorithmic approach is that it assigns a complexity number, or rating, to the code, where the higher the complexity number, the greater the probability of errors. Naturally, the more errors code contains, the more time and effort it takes to maintain and troubleshoot this codebase. Implementing the cyclomatic complexity algorithm on your work will guide you as to where you can improve your code. After establishing any functions with a high complexity value, you can then try to refactor them to reduce their complexity.

Cyclomatic complexity can be applied to functions, methods, classes and modules.

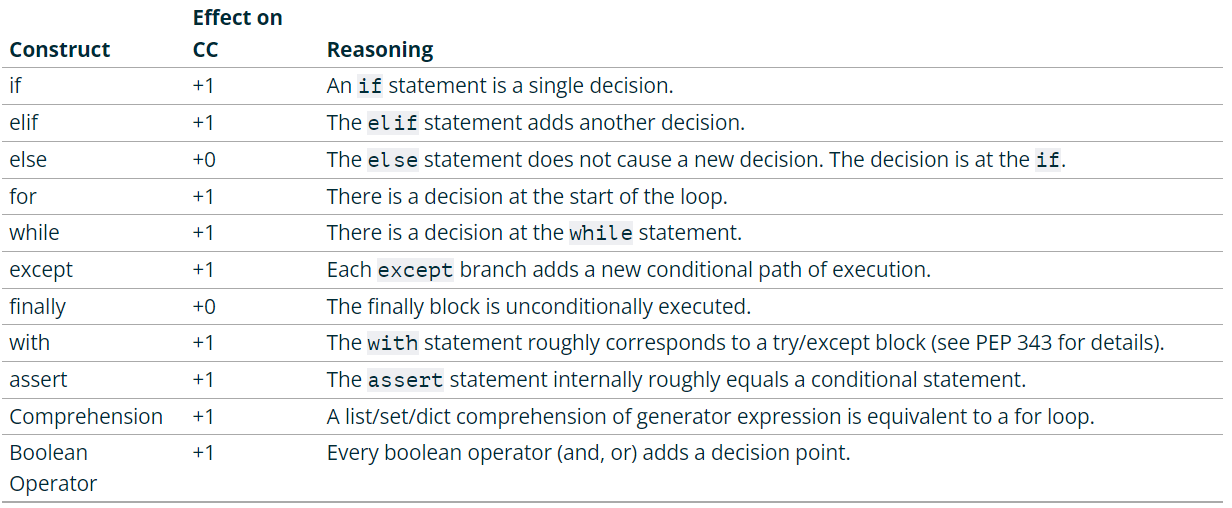


Figure : A table illustrating the effect various constructs have on the cyclomatic complexity number assigned to a section of code. Source: <https://radon.readthedocs.io/en/latest/intro.html>

## Maintainability Index

The last algorithm this report will discuss is the Maintainability Index. One could argue that the logic behind this algorithm is the simplest to understand, especially for somebody who does not come from a technical background. The maintainability index simply measures how maintainable (easy to support and change) source code is, assigning the code an index value between 0 and 100. The higher the index value, the more maintainable the code is. It is that simple. The maintainability index is calculated as a factored formula consisting of SLOC (Source Lines of Code), Cyclomatic Complexity and Halstead volume, all of which have already been discussed in this report. Since the Maintainability Index formula consists of these figures, the first step in using this algorithm is to measure these aforementioned metrics. Once we have these metrics, we can then use the Maintainability Index formula.

The original formula for calculating the Maintainability Index (MI) of code is:



Figure : The original formula for calculating Maintainability Index (MI) of code. Source: <https://radon.readthedocs.io/en/latest/intro.html>

Many companies have adopted this algorithmic metric and some have tweaked the original formula slightly, to get their own preferred variant. One such example is Microsoft Visual Studio 2010, whose altered formula is listed below.

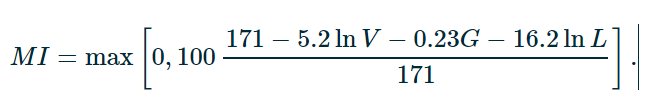


Figure : Microsoft Visual Studio 2010’s altered version of the Maintainability Index formula. Source: <https://radon.readthedocs.io/en/latest/intro.html>

<https://www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity/>

The key for reading these formulas is listed here:

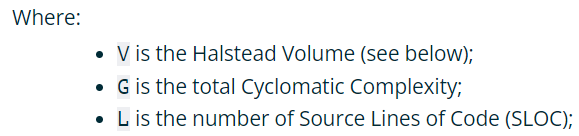


Figure : Key for understanding Maintainability Index formulas listed above. Source: <https://radon.readthedocs.io/en/latest/intro.html>

It is clear from the above formulas that the most influential factor in the calculation of the Maintainability Index is the Source Lines of Code. This is because the coefficient of the Source Lines of Code, 16.2, far exceeds the coefficient of the Halstead’s Volume, of 5.2ln, and of the Cyclomatic Complexity, which is 0.23.

The most important point to take from discussing these algorithmic approaches is not which approach to use, it is to highlight the effectiveness of using any one of these algorithms. The point is, that by utilising any one of, or combination of, these code metrics, developers can understand which parts of their code should be reworked or more thoroughly tested. Development teams can identify potential risks, understand the current state of a project, and track progress during software development (<https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022> ).

# Is This Ethical?

* Moral Issues – Privacy
* Benefits of Measuring Software Engineering
* Do Benefits Outweigh the Cons?

## Privacy

## Benefits of Measuring Software Engineering

The following resource lists some of the benefits:

<https://www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity/>

Remember, what gets measured, gets managed (<https://www.pluralsight.com/product/flow>

).

## Do Benefits Outweigh the Cons?

# Conclusion

Source:

<https://blog.pragmaticengineer.com/can-you-measure-developer-productivity/>

<https://www.gitclear.com/blog/the_4_worst_software_metrics_agitating_developers_in_2019>

<https://www.pluralsight.com/blog/teams/lines-of-code-is-a-worthless-metric--except-when-it-isn-t->

<https://stackify.com/track-software-metrics/>

<https://humanitec.com/blog/lead-time-a-key-metric-in-devops>

<https://www.pluralsight.com/product/flow>

<https://waydev.co/software-engineers/>

<https://codeclimate.com/>

<https://www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity/>

<https://en.wikipedia.org/wiki/Cyclomatic_complexity>

<https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022>

<https://en.wikipedia.org/wiki/Halstead_complexity_measures>

<https://www.javatpoint.com/software-engineering-halsteads-software-metrics>

<https://www.researchgate.net/figure/Halstead-Complexity-Metrics_tbl1_319481865>

<https://radon.readthedocs.io/en/latest/intro.html>

<https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022>

<https://www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity/>