Measuring Software Engineering

REPORT SUBTITLE

Oisin Tong | Measuring Software Engineering | 18323736

Contents

1. Why Measure Software Engineering

Before we examine how software engineering is being measured, it is imperative that we understand why it is being measured in the first place. Without this fundamental basis, how can we objectively compare the utility of any possible metric for software engineering?

Software engineering is measured to do the following three things:

CHARACTERISE

By characterising software engineering, we can gain an understanding of the processes, products, resources, and environments, which are a part of the cycle. In doing so, it enables us to establish baselines which can be used for comparison in future works.

EVALUATE

Evaluating enables us to determine our immediate circumstances with regards to projections. We can decide if our projects are drifting off track, and make changes to rectify our status if so, thus getting things back under control. Evaluation is also a means by which we can assess the quality of our work, and to assess the impacts that changes made in our workflow have had on our work.

PREDICT

Prediction works on the back of characterisation and evaluation. Measuring software engineering provides a greater understanding of the relationships between all the moving pieces, and how they interact and affect one another. This knowledge, based off of past experience, can be used to make predictions for an upcoming project. This empowers us to establish attainable goals to set in terms of cost, schedule, and quality, although accurate estimations in software engineering is notoriously difficult.

With our goals for measuring software engineering, we are now ready to delve into the various methods being implemented in today's culture and weigh the merits of lack thereof which each metric possesses.

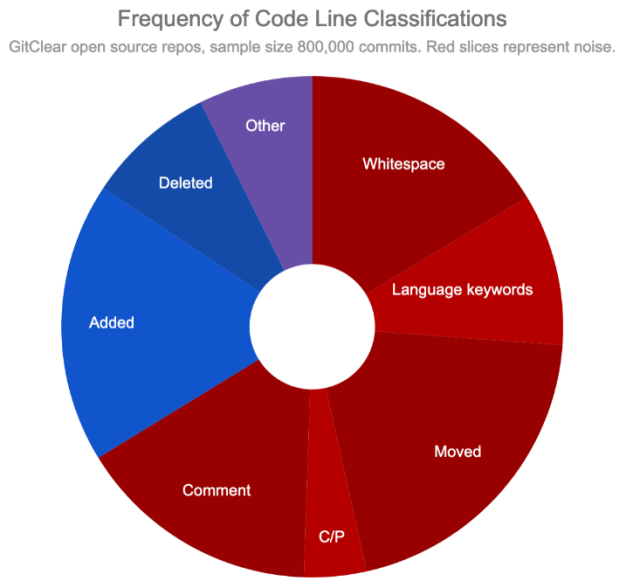
2. Measuring Software Engineering

As stated before, there is no shortage of options available to us for measuring software engineering. In this report, we will examine five different metrics, and examine the flaws and merits to each.

There are a multitude of ways in which the process of software engineering may be measured.

LINES OF CODE

Lines of code is one of the oldest metrics by which software engineering has been measured. At surface value it may appear to offer a tangible sense of how complex and large a project is, but there are quirks to software engineering which may appear initially counter intuitive. First is that the expected amount of code which a project should contain will vary greatly between each project. This is quite unlike in something such as architecture, where buildings of a certain size made of a certain material can be predicted to cost the same amount as one another.
Secondly, the number of lines of code written bear no indication of the quality of said code, and a rule of thumb in software engineering is that performing a task is most desirable when done in as little lines of code as possible.
And so examining lines of code can often make people draw misleading conclusions, to the point where it is largely agreed that lines of code is an unsuitable metric for measuring software engineering.



Pie chart demonstrating how lines of code is rarely an indication of good or bad software engineering

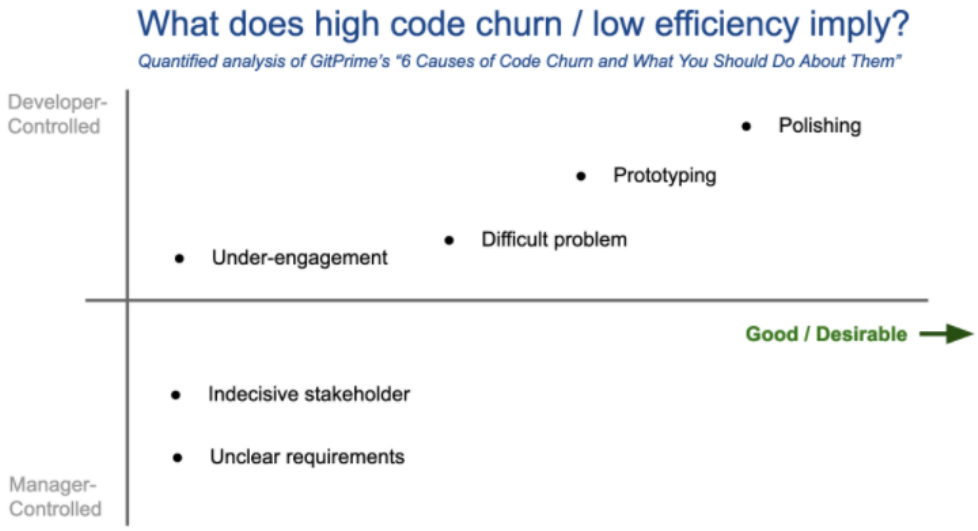
COMMITS

The number of commits made by a team or an individual is a metric similar to lines of code in its contemporary stigma. The premise is that a productive team or engineer will commit code to the git repository more frequently than an unproductive party. However, as with the lines of code, tracking the quantity of commits bear no indication of quality, and indeed, it has been seen in practice that measuring and incentivising a high quantity of commits results in individuals making inconsequential commits for the sake of padding their stats. Thus, this too is a metric which can result in negative implications.

CODE CHURN

Code churn is a measurement of the amount of old code being refactored or removed. The meaning which can be drawn from how much code churn a project contains can vary quite dramatically, but it is nevertheless a powerful indicator for a team to examine.

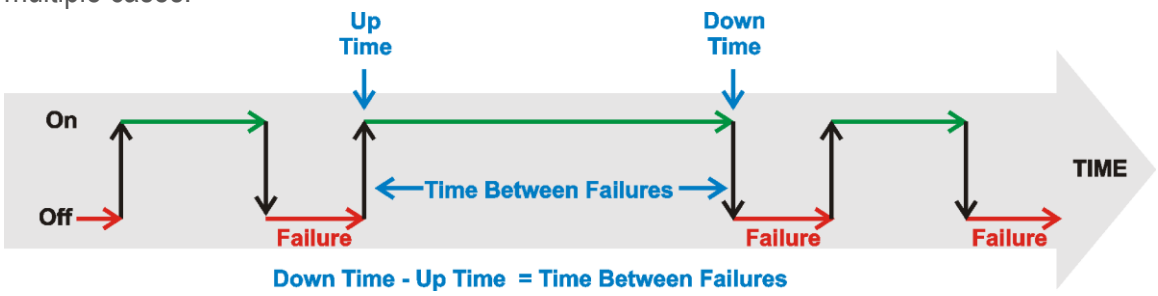
As seen in the diagram below, code churn may be a worrying sign of poorly written code which is requiring constant fixes and work arounds, or it may be a reassuring sign that the team is putting the finishing touches to a successful project.



Graph depicting various causes of code churn, and their implications

MEAN TIME BETWEEN FAILURES (MTBF)

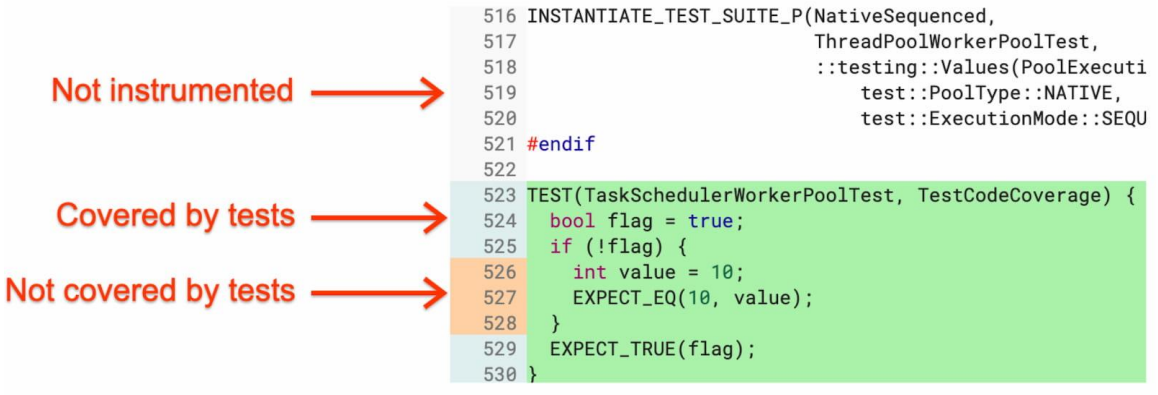
Mean time between failures (MTFB) measures the mean time period which passes between system failures. It is a metric best used during the maintenance of a product when customers are using the service, rather than the iterative and prototyping nature of the initial build. MTFB provides a good indication of the quality of code written for a particular system, as frequent failures suggest shortcomings in the code to deal with recurring and or multiple cases.



Visualisation of MTBF

CODE COVERAGE – AMOUNT OF CODE COVERED BY AUTOMATED TESTS. HIGH CC SUGGESTS BUG-FREE PRODUCT

Code coverage is a measurement of the amount of code which is tested by automated tests. This metric serves a similar purpose as MTBF in that it is a good indication of code quality, although code coverage is a more appropriate measure during the build phase as well as the maintenance phase. A system with little to no code coverage is more prone to bugs and failures. As well as this, in the event of a bug, low code coverage makes it more difficult to identify the exact cause of the problem, resulting in a manual comb-over of the code.



Visual representation of code coverage

3. General Principles

Before examining particular metrics and their value as a means of measuring the software engineering process, there are a number of general principles which should be observed to maximise the usefulness of said metrics.

LINK METRICS TO GOALS

A metric with nothing to assign relative value to it has no use. A successful result should be taken as one which meets pre-defined criteria set by the team. For instance,

TRACK TRENDS

Complex processes such as software engineering cannot be succinctly expressed by numbers taken in isolation. For instance, a struggling team may adopt a change to how they organise themselves, and find at the end of the month that their metrics are still below desired outcomes. However, if they take the trend instead of the single data point, they may find that despite failing to reach their target, that their change in organisation has in fact resulted in a significant improvement over where they were previously.

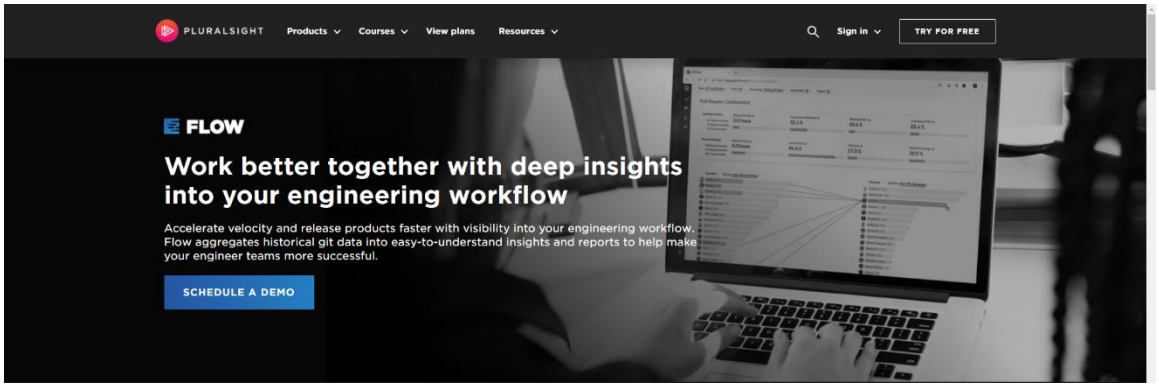
SET SHORT PERIODS OF MEASUREMENT

Using many short time samples instead of few large time samples has a number of benefits. First, it provides more opportunities for teams to identify potential problems in advance and adapt accordingly. Secondly, having more data points allows for a more comprehensive and accurate representation of what is going on in your project.

4. Available Computational Platforms

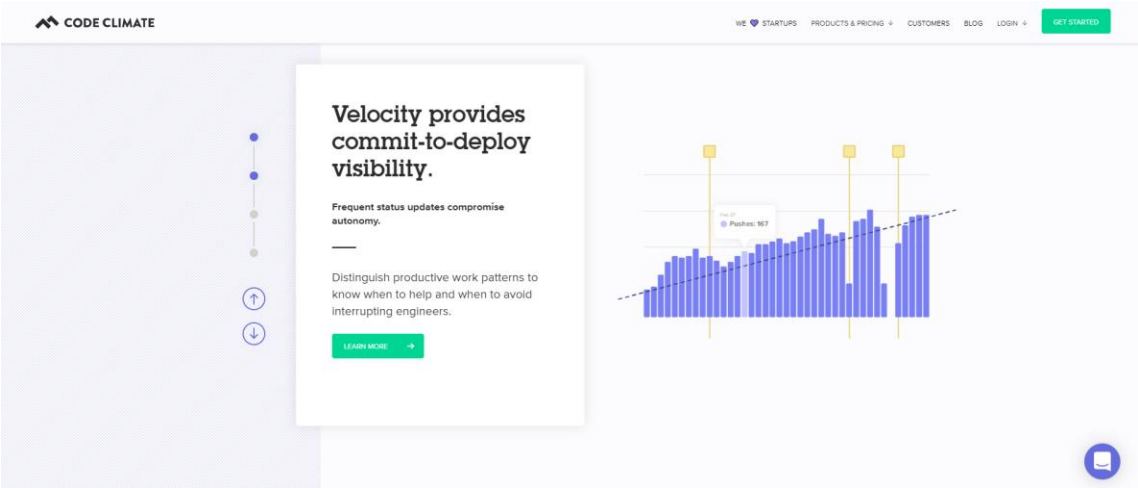
All of the most prominent computational platforms currently available make use of the client's git repositories, such as Github, to aggregate its data and formulate insights and visualisations for the client to digest. Each of these platforms are also incredibly simple to set up and get started with, and provide ample documentation to enable developers to make the most of their subscriptions. Of the options available on the market today, some of the more notable platforms for measuring software engineering include the following:

PLURALSIGHT – FLOW



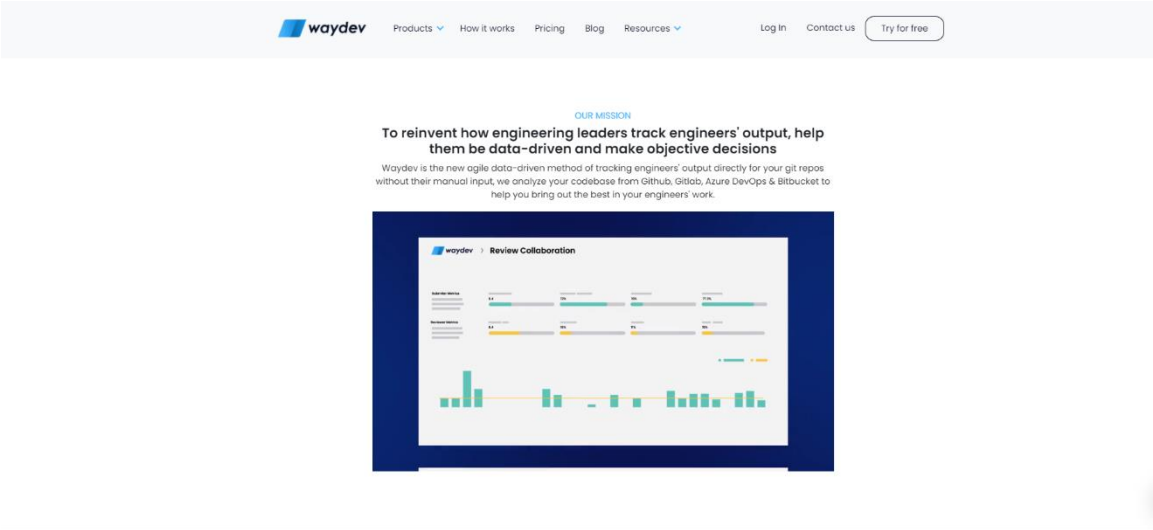
Flow is a tool originally known as GitPrime before it was acquired by Pluralsight in 2019. Flow uses the data extracted from the client's collection of repositories to return a wide range of statistics. One such metric of note is a comparison between the time spent by engineers refactoring code instead of writing new code. Of course, as we discussed in the previous section, code churn can be caused by many different sources, but this is valuable information nevertheless because a team can look at this data and interpret the results based on their current circumstances. For instance, if a team is close to the end of a project timeline, finding that a lot of their time is being spent writing new code instead of polishing existing code could be an indication that they are behind in terms of implemented features, or perhaps that they are being too ambitious and ought to instead ensure stability of the product rather than adding new potential sources for bugs.

CODE CLIMATE – VELOCITY



Another tool which utilizes the client's git repositories, Velocity advertises an interesting metric for software engineers; it graphs the commits of individual engineers over time. Despite having established that counting commits is a flawed approach towards evaluating a software engineer, Velocity use of such a metric is fascinating in that it uses it for quite an abstract notion. Instead of evaluating an engineers worth, Velocity suggests this metric be used as a means of marking times in which particular engineers are routinely committing the most, and are perhaps times in which a manager ought to avoid interrupting them lest they disrupt the engineers productivity.

WAYDEV



Waydev separates itself from its competitors by marketing itself as an AGILE-focused computing platform. One interesting feature which Waydev offers is a forecast model to be used in tangent with the client's sprints which can help to identify potential scope creep and work overloads given any individual team.

5. Algorithmic Approaches

There are a plethora of algorithms available for computing a wide manner of software engineering metrics. As has previously been established, these algorithms are insufficient on their own to drive decision-making, but nevertheless they are integral in drawing objective data upon which teams can inform their decisions.

Below are some such algorithmic approaches.

CYCLOMATIC COMPLEXITY

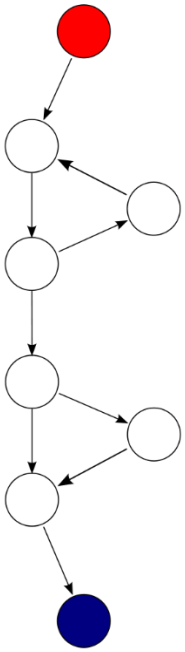
Cyclomatic complexity is defined as the amount of linearly independent paths withing a section of code. For instance, a piece of code without any conditional statements has a cyclomatic complexity of 1, because no matter the circumstances under which it runs, the code can only be executed in one linear path. A single conditional if statement will result in cyclomatic complexity of 2, as two unique paths are formed depending on whether the if statement returns a TRUE or FALSE.

The algorithm for calculating cyclomatic complexity is given as:

M = E – N + 2P

Where:

M = Cyclomatic Complexity
E = the number of edges of the graph.
N = the number of nodes of the graph.
P = the number of connected components.



A graph representation of a program with cyclomatic complexity of 9 nodes, 8 edges, and 1 connected component = Cyclomatic complexity of: 9 - 8 + 2*1 = 3

MEAN TIME BETWEEN FAILURES

MTBF is a relatively simple metric to define as an algorithm.
It can be written as such:

$$\text{MTBF} = \frac{\sum (\text{start of downtime} - \text{start of uptime})}{\text{number of failures}}$$

CODE CHURN

Code churn is another metric which is theoretically simple to translate into an actionable algorithm. It can be defined as:

$$\frac{Lines\ of\ code\ refactored}{Lines\ of\ code\ written}$$

This will return the percentage of code which has been refactored and will give an understanding of a project's code churn.

6. Ethical Concerns

As with all things, I believe that it is the implementation which dictates whether a tool or invention is ethically immoral. Having said that, however, it's hard to argue that certain things lend themselves to unethical behaviour more than others. In the case of the measurement of software engineering, and of the engineers themselves, I believe it may very well better enable teams or individuals to identify their shortcomings and address them in a thorough and direct manner. Despite this, however, its seems to me an inevitability that such tools of measurement will be increasingly used as a means to intrude on individuals privacy in an attempt to squeeze as much work out of them as possible.
This can be seen today outside of the field of software engineering. Only days ago, controversy has arisen as a result of Microsoft's 'productivity score' which allows managers to monitor individuals usage of the Office 365 suite. Similar monitoring may be implemented for software engineers, and I think in this regard it is safe to say that the opportunities this reveals for self-improvement pales in comparison to the potential damage done to individual's privacy and mental well-being.

7. Conclusion

The pitfalls and dangers of measuring software engineering are many, and there is a fine line which must be tread between measuring performance with the aim of improving upon it, and breaching individuals right to privacy.
In this regard, the field of software engineering is in a fortunate position in that not only are its engineers responsible for creating their own methods of self-measurement, but they can also see how current iterations of productivity measurement has affected workers of other disciplines.
I believe this goal of measuring software engineering shares many traits with the pursuit in AI in that simply because something can be done does not mean that we are yet prepared for the implications of going down such a path.

8. Sources

https://www.youtube.com/watch?v=cRJZldsHS3c&ab_channel=GitHub

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

https://stackify.com/track-software-metrics/#:~:text=Managers%20can%20use%20software%20metrics,problems%20within%20software%20development%20projects.

https://resources.sei.cmu.edu/asset_files/Handbook/1996_002_001_16436.pdf

https://www.brooksinstrument.com/en/blog/understanding-mean-time-between-failure-mtbf-for-process-instrumentation

https://developerexperience.io/practices/code-coverage