Measuring

Software Engineering

Report

Author: Samuel Alarco Cantos

# Introduction

Software engineering has become one of the most dominant and relevant engineering disciplines of the modern era. Traditional industries such as transportation, communication, entertainment, and hospitality are gradually being transformed or outright disrupted by the software revolution. As software engineering is ingrained into many companies and processes, becoming a vital part of countless businesses and enterprises, a simple but profound question has been raised: can we measure it?

It is the ubiquitous temptation of entrepreneurs, managers, investors, and team-leads alike: can we measure the performance of a team, quantify quality of a product, calculate the efficiency of an employee. The success of the modern business and managerial models are arguably based on the usage of useful and actionable metrics. As software engineering quickly develops and becomes such an important discipline, it is inevitable that the same questions are asked. Even software engineers themselves are usually interested in their own performance.

However, is this possible? Are these useful questions to ask? Can they even be answered with any degree of accuracy or actionability? What are the ethical consequences of measuring software engineers as individuals and teams with the data and methods we have available? What conclusions can we draw from these measurements and how should we act on them? On this report, I will explore these questions and try to spark a discussion on the most important topics of interest.

# Measuring Engineering

In this section I will explore what is meant when talking about measuring software engineering. I will discuss several aspects and dimensions that could be measured, together with the motivation and relevance for such measurements.

## Measuring Code

The first aspect that comes to mind when measuring a process such as software engineering is measuring the outcome or product of the process. This can be thought of in two ways: the end-product that is the software shipped to the client, and the source code which is the raw artifact created by the software engineering process. The multiple ways of measuring the efficiency, impact, and success of a product in the market have been thoroughly researched and discussed in literature. I will not dwell in them since they are also dependent on extraneous factor such as business planning, product design, marketing, etc. which are outside the scope of this report. The analysis of the source code is of much more interest to the measurement of the software engineering in this case, even if it has many limitations.

When thinking on how to measure software engineering through the source code generated, several metrics quickly come to mind. We could measure things such as lines of code written, or changed by an engineer, quality of the code, idiomaticity, or complexity. Some of these, such as the lines of code per commit, are readily available from version control systems such as git. Others, such as code complexity, can be calculated through certain mathematical methods such as cyclomatic complexity, even if the accuracy is arguable.[[1]](#footnote-1) In general, these metrics are easy to calculate and can give a general picture on the work of a software engineer or team. However, they are rather useless without further context. Different programming languages, because of their syntax and idiomatic practices, are more or less verbose and have different average line counts. These can further vary due to specific company style guides and best practices. Complexity can be difficult to properly quantify, and often depends on a balance between efficiency, readability, and functionality.

Finally, these measures are easy to game. If a developer is being measured by lines of code written, they will write more verbose and possibly inefficient code. If they are being judged by the number of commits, they will just make more micro-commits. In general, using these metrics as a main source of evaluation for software engineers has unexpected and undesired effects, rendering them useless by themselves beyond giving some nice statistics about the work of a developer.

## Measuring the Process

It is far more useful to measure the software engineering as a process, not just as its end-product. By measuring the process through a wide variety of metrics, we can identify strengths and bottlenecks at individual, team, and inter-team level, that are often resource sinks. All together, they can give us crucial actionable insights into the performance of engineers and engineering teams.

The software engineering process can be framed in terms of **cycle time.[[2]](#footnote-2)** This is a term that comes from the manufacturing world. When referring to code and specifically to software features, it represents the time it takes for a feature to go from design/inception, through implementation, code reviews, testing, integration, to the moment it is released into production. Each of these steps have their own sub-metrics which give further insight into performance of the team. The cycle time can provide an actionable metric describing how well an organization is working together, of if there are bottlenecks.

Several seminal studies into these methods were carried out by DORA, the DevOps Research and Assessment team. They were acquired by Google in 2018 and have as their goal “to understand the practices, processes, and capabilities that enable teams to achieve high performance in software and value delivery.”[[3]](#footnote-3) In the book *Accelerate*, they expounded 4 metrics which they claim can measure the performance and delivery capabilities of software engineering teams. These are:

1. **Lead Time:** The time it takes for a feature or change to go from inception to deployment. This is a close parallel to the cycle time mentioned above. Slow CLTs indicate bottlenecks and inefficient processes in a team or individual.
2. **Deployment Frequency:** How often are new versions deployed in production. This is especially relevant in the context of agile programming, as it indicates how fast software is being developed and (hopefully) improved. DORA suggests that high deployment frequencies indicate fast development with small changes pushed often to production. This is generally considered safer and better practice than slower, bigger deployments.
3. **Change Failure Rate:** How often deployments of changes and new features lead to incidents. It is useful to indicate the quality of the code being produced and pushed to production. It also shows the quality of the testing pipeline being used before deploying code. The aim should always be to reduce this metric.
4. **Mean Time to Restore:** Complementing CFR, MTTR measures how long it takes to recover from a failure. This can be business critical as failure can lead to partial or total blackouts, leading to economic losses and customer dissatisfactions. As with CFR, MTTR should always be minimized.[[4]](#footnote-4)

1. <https://www.perforce.com/blog/qac/what-cyclomatic-complexity>, accessed 20/12/2021 [↑](#footnote-ref-1)
2. <https://www.swarmia.com/blog/measuring-software-development-productivity>, accessed 20/12/2021. [↑](#footnote-ref-2)
3. <https://jellyfish.co/blog/dora-metrics-101>, accessed 20/12/2021 [↑](#footnote-ref-3)
4. *Accelerate, The Science of DevOps,* [Nicole Forsgren](https://es1lib.org/g/Nicole%20Forsgren)*,*[Jez Humble](https://es1lib.org/g/Jez%20Humble)*,*[Gene Kim](https://es1lib.org/g/Gene%20Kim), 2018. [↑](#footnote-ref-4)