Measuring Software Engineering

To deliver a report that considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethics concerns surrounding this kind of analytics.

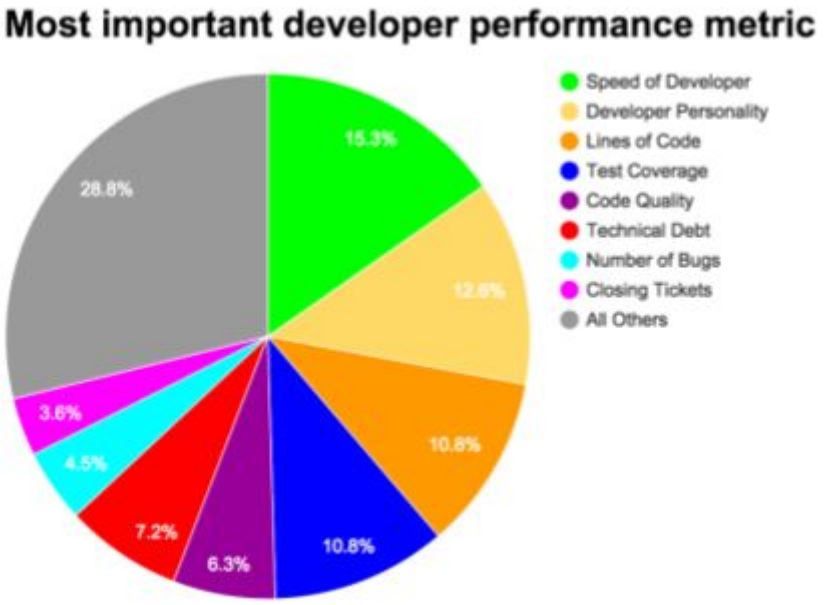
Table of Contents

Abstract

The software engineering process is like no other engineering process in the world today. Unlike other disciplines such as engineering and mathematics there is no defined procedure for a software engineer to complete a task. Every engineer is different and in some ways that is what makes the process so fascinating and especially interesting when we gauge the metrics of the process. Barrett (2017) acknowledged in one of his lectures that "pretty profound things can be uncovered from pretty mundane data". What I will look into in this paper is what data is measurable, where this data is computed, what algorithmic approaches are available and the ethical concerns surrounding the analytics of software engineering.

1. Measurable Data

"There are two kinds of measures in engineering: product metrics and process metrics. The former describes product's qualities such as dimensions, physical data, etc; the latter describes process' qualities such as effort required, production time, etc. In software, the former includes code length, complexity, reusability, maintainability, etc. The latter includes editing time, number and type of changes in a class or in a file, etc." (Sillitti et al., 2003). Whether engineers like it or not measuring data is a necessary practice for their profession. It gives them something to work with when they are estimating the time it may take to complete a job. Thus, using past experiences as benchmarks for completing new jobs acts as a benchmark for future jobs. In this section I will talk about some of the most prevalent metrics used in the industry today.



Source: York, 2015

1.1. Speed of Developer

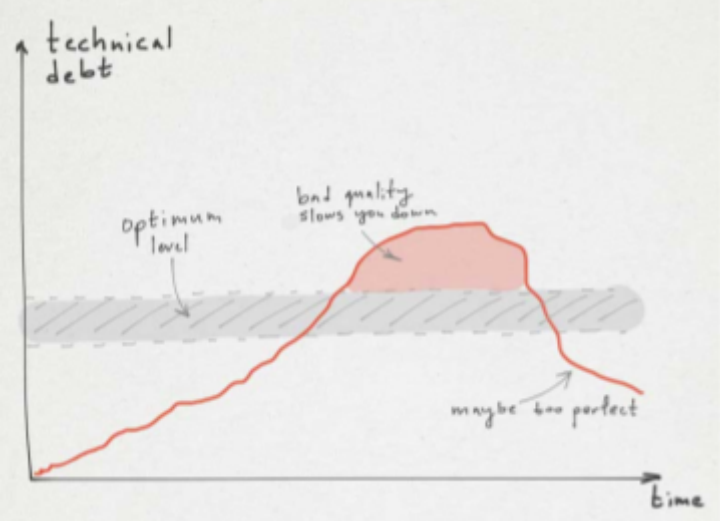
As the saying goes, time is money. Therefore, measuring speed of development should be another way to measure productivity. After all, the point of software development is to deliver working software. The faster that a team delivers, the better. A fast development process also allows for the release of software more quickly and efficiently. But speed will only be the most important metric for certain business models. For others, following the specifications would be of greater importance. On the whole I would consider speed to be a valuable measure of data in the the software process but like its counterparts it does have its flaws. One of them would be that having a development team measured on the speed of the job consistently, will lead to burnout. All in all, it depends on the type of software you write for how beneficial speed of development is as a metric to you.

1.2. Source lines of code (SLOC)

Bill Gates once said “Measuring programming progress by lines of code is like measuring aircraft building progress by weight”. Is this type of measurement really such a bad way of measuring an engineer. Detractors like Gates will say that using this as a metric will encourage bad practices such as copying and pasting code and filling programs with meaningless comments. When trying to look at SLOC as a positive metric I try to think of how non-technical people would see it as an easy metric to game. Length is a measure that they would understand so SLOC can be a good way of showing someone the work of a project. Still, I believe SLOC is not the best way to measure a software engineer. So many analogies like Gates’ above have been used to show the ineffectiveness of this methodology and most in the software engineering world do not like this method. This metric does a very poor job in measuring programming progress and developers productivity. A programmer who is being measured in SLOC, will be rewarded for generating more lines of code rather than code that works. How could this be considered best practice.

1.3. Technical Debt

Technical Debt is a deliberate decision to implement a flawed solution or a lower standard of code in order to release software faster. Doesn't sound like the best policy but in some cases technical debt can be acceptable. It is impractical to have zero technical debt as it will lead to the slow delivery of new features. As a means of measuring the software engineering process technical debt is becoming increasingly popular. Software will rarely be perfect so treating technical debt as an investment in the future of the software life cycle is becoming the norm. Reasonable levels of technical debt is now considered good practice because it allows new features and products to be released without having to revisit old bugs regularly. Of course, if these bugs need to be fixed then the debt will need to be repaid at some stage.



Source: Dubakov, 2014

1.4. Communications

Unlike the metrics talked about above, communications isn't necessarily a quantitative metric. This metric is non-code based but in many ways is just as valuable as some of the metrics discussed above. Software engineers, like other knowledge-workers, spend a ton of time collaborating across instant messaging, email, reading designs, reviewing code and going to meetings. It should be a priority for companies to hire software engineers with great communication skills. Experienced software engineers such as Flood (2015) have even gone as far as saying that "all things being equal, the better you are at communication the further your software engineering career will advance and it will do so a faster pace." It is difficult to measure how good someone is at communicating compared to how many lines of code they write per day but the better the communications between software engineers and their clients the better the end solution. Thus, a high level of communications is an extremely useful piece of data to analyse when measuring an engineer's performance.

2. Computational Platforms Overview

Computational platforms are basically where we compute the types of data measured from the software engineering process. In this section I will talk about the different types of code review platforms; why they're used and why not and also the growing trend of Analytics as a Service.

2.1. Analytics as a Service

AaaS refers to the practice of using web-based technologies to carry out analysis of big data, opposed to the traditional method of developing an onsite hardware warehouse to collect, store, and analyse the data. The new set-up that AaaS provides allows clients to use a particular analytics software for as long as needed, and can be more cost effective and less labour intensive when compared to the traditional way of building the necessary infrastructure in-house. Outsourcing the bulk of this work to reduce cost is becoming an increasingly popular option but hybrid options like using tailored hardware combined with web-based infrastructure is allowing companies to create even more suitable solutions for analysing their big data needs.

2.2. Code Review Platforms

Analysing the measurable data talked about in section 1 has become far easier nowadays with the introduction of code review platforms. Of course, communications between software engineers cannot be measured but data like technical debt, lines of code and maintainability are types of data that code review platforms can pick up on. The three platforms that I will compare and contrast below are code climate, scrutinizer and codacy.

2.2.1. Code Climate

Pros:

- a great number of supported languages, technologies and frameworks,
- nice User Interface
- well-maintained test coverage feature gem
- trends charts
- test coverage out of the box

Cons:

- seemingly an integrated bunch of open-source projects
- pricing - it seems to be the most expensive tool in this comparison
- still an unpredictable API
- no detailed description of the issue, only a header with source code
- no interest from the Code Climate team to extend the tool the way customer may suggest.

Source: Ozimek, 2017

2.2.2. Scrutinizer

Scrutinizer doesn't stack up well against other solutions; lacking the basic attributes that are necessary in code review.

Pros:

- seems to have very good API
- it's actually one of the cheapest solutions if we ignore the performance
- automatically detects code changes in the dashboard
- well written documentation
- a feature of filtering issues by users
- dedicated site with current status of the services

Cons:

- dead test coverage gem (only 2 commits 3 years ago),
- API available only when subscribed to the most expensive package,
- not a predictable tool.
- performance: you pay per container and a container is simply *one task that can be run at once*, fewer containers - less performance and longer waiting time for your code to be analyzed,
- there's no option to see all the rules (checks), there are two groups for enabled or disabled rules, but there's no "all" option.

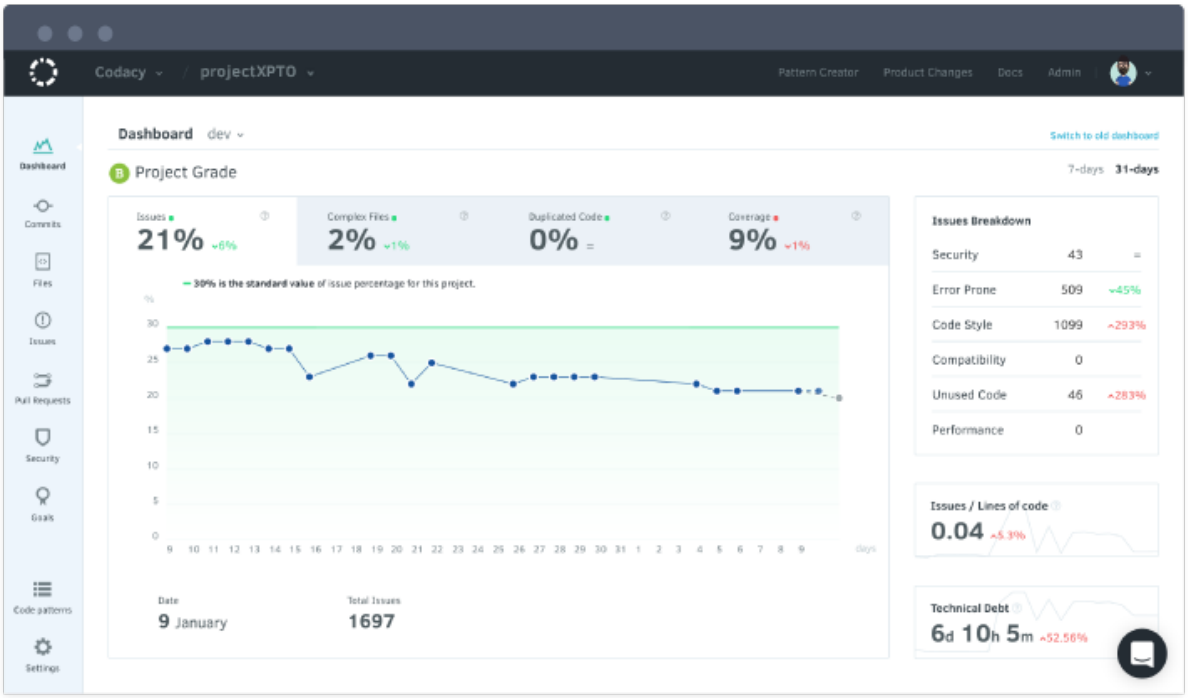
Source: Ozimek, 2017

2.2.3. Codacy

Pros:

- great and intuitive UI
- The dashboard is clean and really easy to use (see screenshot below)
- For each project, Codacy provided metrics on the quality of my project.
- Each piece of information is clearly reported and each file receives a score based on a letter from A to F.
- a nice feature of browsing commits and monitoring related issues
- time to fix estimation for each issue
- well-described issues with examples right below each case for a swift fix

Source: Leroy, 2017



Source: Codacy

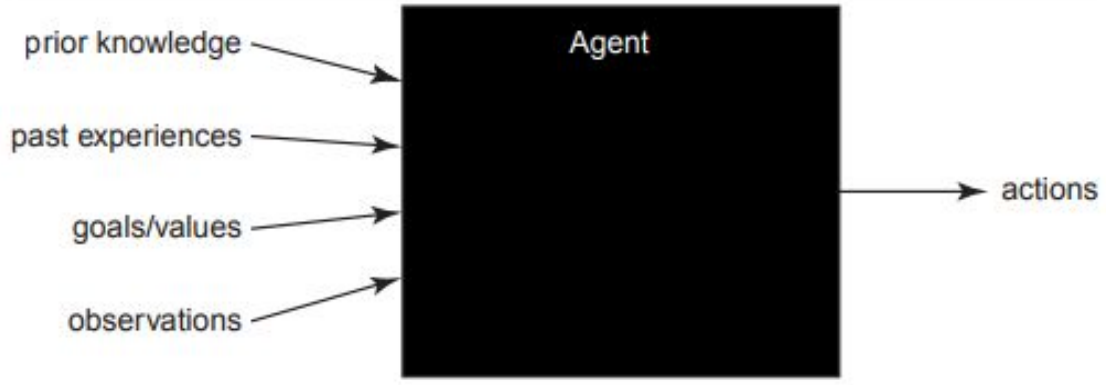
Of the three platforms discusses above it is hard to argue that the service they provide to companies is invaluable. They help to improve existing workers performance while also putting pressure on any employees who may be underperforming in the fairest manner possible. If I were to choose between one I would probably choose Codacy. Just from analysing it with my own Github it showed me some of the cool features highlighted above. I immediately saw the use and benefits of this platform. Couple this with the fact some of the big players like PayPal and Adobe are using it and it is easy to see why Codacy is becoming one of the best code review platforms available.

3. Algorithmic Approaches

Much of software practice centers around daily decisions and questions (e.g., when to release a software system? Unfortunately, nowadays many decisions related to a software system are based on intuition and gut feeling. These hastily made decision processes lead to wasted resources and increased cost of building and maintaining large complex software systems. Software Intelligence (SI) intends to put these issues to bed. Just like Business Intelligence uses statistical models to make accurate forecasts and improve decision making in the business landscape, software intelligence is starting to be implemented in a similar fashion. SI offers software practitioners (not just developers) up-to-date and pertinent information to support their daily decision-making processes (Hassan & Xie, 2010).
Below, I will look at how these practices are being implemented in terms of computational intelligence, machine learning and decision making algorithms.

3.1. Computational Intelligence

Computational Intelligence (CI) is the study of the design of intelligent agents. An agent is something that acts in an environment—it does something. In this case CI refers to the ability of a computer to learn a specific task from data or experimental observation. The goal of computational intelligence is to understand the principles that make intelligent behavior possible, in natural or artificial systems. Systems designed using CI are flexible to changing environments and changing goals, learning from past experiences and adapting dynamically to solve particular problems.



Source: Poole et al., 1998

The diagram above depicts a computationally intelligent system as a black box. Success in building an intelligent agent naturally depends on the problem that one selects to investigate and the contributing factors of knowledge pumped into an intelligent system to create an end product that solves the problem.

3.2. Machine Learning Algorithms

Machine learning is the science of getting computers to act without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition and effective web search. Machine learning is so pervasive today that we use it dozens of times a day without realising it (Ng, 2017). Below I will talk about two types of machine learning algorithms employed by software engineers to influence the world around us.

3.2.1. Supervised Learning

This is the most common type of machine learning process. The term supervised learning refers to the fact that we give the algorithm a data set in which the "right answers" (output variables) are given. The goal is to approximate a mapping function so well that when you have new input data that you can predict the output variables for that data with a supervised learning algorithm. This is useful for predicting what will happen when a certain set of circumstances arise in the real world that have already happened in the past.

3.2.2. Unsupervised Learning

Unsupervised learning is where you only have input data and no corresponding output variables. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data. These are called unsupervised learning because unlike supervised learning above there is no correct answers and there is no teacher. Algorithms are left to their own devises to discover and present the interesting structure in the data. Unsupervised learning algorithms use the technique of clustering to make use of the input data presented to them. An example of this would be Google News and how they group their information. What Google News has done is look for tens of thousands of news stories and automatically cluster them together based on keywords in the headlines and in the story. Then, once the news stories are clustered appropriately, stories about the same topic get displayed together on search screens and the home tab.

3.2.3. Algorithmic Decision Making

Algorithms — quite rapidly and without debate — have come to replace humans in making decisions that affect many aspects of our lives. Algorithms that most of us wouldn't think of affect us on a daily basis. We rarely realize how much blind trust we place into algorithmic decision-making. Where I think the use of decision making algorithms could be taken advantage would be to eliminate the potential for human bias in decision making. For instance, Criminal sentencing is one area in which algorithms are theoretically useful, because they can help eliminate human racial bias in the criminal justice system. Also, algorithms don't think when they are approving credit cards, counting votes, or determining financial aid. Therefore, assuming that we provide data to the machines with high integrity

they should be able to make a fair decision from a tailored supervised algorithm for the situation.

4. Ethical Concerns

One of the dangers of gathering all this data is that are we really allowed have this available for us to use at any given time. It must be considered that data gathered from the software engineering process allows us to answer questions that maybe we should not even be allowed ask. In my opinion, the most concerning part of software engineering is what we could do with the data generated from the process. We place so much trust in people we don't even know not to take advantage of the sensitive data we provide to them on a daily basis. Chacko (2016) talks about how "the role of a software engineer is not limited to designing, developing and deploying software projects. He/she should also possess sound knowledge and perseverance in all such ethical concerns without losing the scope and quality of products that are delivered".

One major ethical concern of software engineering is privacy. None of the software projects should cause any hindrance to privacy of individuals or organizations. This is another major ethical concern that most of the software engineering firms have in addition to all others. Another concern for companies would be social implications and cyber crimes. The involvement of the public in all major social software projects is appreciable and talented software engineers relentlessly work hard to accomplish the same. They are also concerned about the cyber crimes that can take birth after the deployment of their software.

4.1. Legal Limits

Here, I will talk about three factors that need to be considered so that software engineering projects are legally feasible. These are privacy, intellectual property and liability.

4.1.1. Privacy

The core principles of privacy are: 1) freedom from intrusion, 2) control of personal information, and 3) freedom from surveillance. With the increase in information technology has come an increase in privacy concerns. Privacy is not clearly defined as a right in the Irish constitution but actors privy to sensitive data usually are restricted to exposing this data and thus the courts recognise that the personal rights in the constitution imply the right to privacy. I believe that as software engineers become more and more exposed to a wealth of knowledge in their line of work they will need to be increasingly monitored to comply with the privacy of others.

4.1.2. Intellectual Property/ Data Sovereignty

Who owns the data? Take for example a Facebook post from someone's private account. Who owns the rights to the data generated from this. Is it Facebook or is it the user and

why does this matter. Well, with Facebook the lines are kind of blurred. As they have created a service that you have decided to use they can do whatever they want with what you provide them but you still own any of the material you post. Facebook can do whatever they wish to do so just like any other company. This would be the same for any other company you enter into a contract with. They can argue what you provide to them is theirs to peruse when they feel like doing so. However, companies are wary of crossing any lines when dealing with this information as they could face public backlash from releasing the kind of information they possess. In my opinion, there will always be a tension between what companies can do according to its policy, and what they will do.

4.2. Transparency of Data Measurement

In my opinion, transparency of how we acquire and use the data is key when we talk about the ethics concerns surrounding this kind of analytics. Transparency facilitates accountability - verifying that services perform as they should, and that data is used according to contract. Fortunately new methods to increase the transparency of data measurement are being introduced more and more. These approaches include:

- personal information management
- program verification
- provenance and distributed access control

I believe that once we are all held accountable in this area of transparency, from the software engineers to the multi national companies then the overall best practices will improve and we will all be more accountable for how we use data and what we use it for.

4.3. Beneficial or Detrimental

In general I feel that any sort of monitor on a worker is a good thing. Without measurement there can be no measurable improvement. Therefore, for the software engineers of this world I think the measurement of their performance makes better software engineers out of them. Upholding a high ethical standard is also of importance to the product created. With good ethical practices in place, teams will be formed with high integrity and competence.

This in turn creates better products for consumers and clients to use and a better software engineering environment; from the developer to the user. Looking at the detriments of using the data generated from this process would be mainly the privacy issues the process could potentially violate. However only a small proportion of engineers would ever cross this line in their lifetime. Thus, on the whole using these kind of analytics is generally beneficial to society as the improvements/ results we generate from using them generally outweigh the negatives.

Bibliography

Barrett, S. (2017). *Measuring Software Engineering*.

Chacko, T. (2016). *Ethical concerns in software engineering that the world rarely discuss*. [online] Linkedin. Available at: https://www.linkedin.com/pulse/ethical-concerns-software-engineering-world-rarely-discuss-chacko/ [Accessed 28 Nov. 2017].

Dubakov, M. (2014). *Speed in Software Development.*. [online] Targetprocess. Available at: https://www.targetprocess.com/articles/speed-in-software-development/ [Accessed 23 Nov. 2017].

Flood, D. (2015). *Communication is Key to Software Engineering*. [online] Linkedin. Available at: https://www.linkedin.com/pulse/communication-key-software-engineering-donnie-flood [Accessed 25 Nov. 2017].

Hassan, A.E. and Xie, T., 2010, November. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 161-166). ACM.

Leroy, S. (2017). *Codacy: A New Easy-to-Use Code Quality and Review Automation Solution*. [online] dzone.com. Available at: https://dzone.com/articles/codacy-a-new-easy-to-use-code-quality-and-review-a [Accessed 28 Nov. 2017].

Ng, A. (2017). *Unsupervised Learning*. [online] Coursera. Available at: https://www.coursera.org/learn/machine-learning/lecture/olRZo/unsupervised-learning [Accessed 28 Nov. 2017].

Ozimek, Ł. (2017). *Comparison of Automated Code Review Tools: Codebeat, Codacy, Codeclimate and Scrutinizer*. [online] Netguru. Available at: https://www.netguru.co/blog/comparison-automated-code-review-tools-codebeat-codacy-codeclimate-scrutinizer [Accessed 28 Nov. 2017].

Poole, D., Mackworth, A. and Goebel, R., 1998. Computational intelligence: a logical approach.

Sillitti, A., Janes, A., Succi, G. and Vernazza, T. (2003). *Proceedings of the 29th Euromicro Conference.* Los Alamitos, Calif.: IEEE Computer Society, pp.336-342.

York, B. (2015). *The Best Developer Performance Metrics*. [online] Medium. Available at: https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0 [Accessed 23 Nov. 2017].