Measuring Software Engineering

By Leon Sinclair

15320877

This essay will be discussing the difficulties associated with attempting to measure the work of software engineers, how it might be done, and what can be gained from doing it.

Diving right in, why is it hard to measure software engineers productivity? The first problem is defining productivity, many of the immediate answers are very easy to game (as discussed later) and effectively worthless. Many other measures are unfortunately only partial measures that apply in certain circumstances and can't generate a complete picture of performance. Simple assembly-line style measures don't capture the nuances of software development and generally lead to people writing poorer code to try game the system. So, what are our options?

What Data Should Be Collected?

The Basic Tier - Easy to Game

Number of Commits, Lines of Code, Features delivered, Sprint Targets Met, Test Coverage, Keystrokes, Estimates, Meeting Lengths.

Several simple metrics exist for measuring productivity, however they all tend to have similar flaws in that they are very easy to game. It is quite simple to expand code and inflate the number of lines it takes which can lead to overly verbose and bloated code. Good code is also often short, but considering shorter code a goal can lead to incomprehensible one-liners that are impossible to maintain. In general these metrics are easy to game and gaming them often leads to a drop in quality of code and process. If a team is measured on how short their meetings are they might rush through them at the expense of proper discussion of issues and features. Features delivered suffers similarly as new ones are rushed out the door to increase throughput.

Estimation of deadlines and meeting sprint targets all have issues with the fact that they rely on the engineer being perfectly honest about their own productivity. As soon as you add weight to them being underestimate then you create an incentive to be dishonest and deliberately overestimate. This makes the metric utterly useless unless you start twisting how it works and doubting your engineers but then you waste time questioning deciding if estimates are reasonable or not.

The fundamental problem with these metrics is that engineers tend to be very good at solving problems and getting the best output of systems. Unfortunately, this also includes systems for measuring them.

Advanced Tier - Harder to Cheat but Harder to Use

Quality Commits, Tests Coverage (tests written by other devs), Number of bugs found during reviews, Code Quality, Cyclic Complexity, Code performance - speed or uptime, Time spent fixing bugs, Fix rate, Idea Lead Time, Cycle Time.

These metrics aren't quite as simple as before but they aren't anywhere as easy to cheat. For example having another engineer review code and note how many bugs they find is a simple way to a) improve software quality and b) remove the issue of someone creating bugs for them to find as finding bugs doesn't improve their own metrics. Code Quality is mostly measured by external services (such as Codacy - discussed later) and it can be useful to see if developers are producing shoddy or error-prone code. Since these metrics tend to involve external analysis, they aren't as easy to game. Others such as cycle time (time from start of work to product delivery) and lead time (time from idea being requested to product delivery)

are good for measuring how well the team and process work at finishing tasks and delivering features/bug fixes.

Time spent fixing bugs and fix rate are interesting measures, it's important to note that spending a lot of time fixing bugs might not be a symptom of a poor developer. It may be that there is a lot of technical debt and poor legacy code that need large amounts of maintenance, an engineer spending a lot of time fixing that could be considered quite a valuable asset! However if the bugs are from newer code and seem to pop up frequently then this may be a sign that poor code is being produced currently and the original author of the code needs to be looked into.

Cyclic Complexity is a somewhat unreliable measure but it can point out sections of code and flag them as areas of concern. It is possible that this level of complexity is necessary or unavoidable to write good code but it is important to check and make sure that this is the case.

Overall, these measures are nowhere as easy to game but are normally more situational, rather than being hard and fast boxes that need to be checked off they should be used as flags and indicators of potential issues/poor performance.

Zooming out from the individual.

We can also gain information about individuals by looking at the performance of the team they're working in. If someone's presence coincides with lowered productivity then it is possible that they are producing poor code that is drawing attention away from other work. This one is very tricky though as it can lead to issues with people either carrying weaker programmers or being carried and hiding their deficiencies. It also creates a negative atmosphere as less productive members will be viewed as "dragging down" better engineers and making them look worse. It may not be the fault of the newcomer that productivity has dropped however, for example others in their team might dislike them (through no fault of the newcomer) and this friction is leading to less productive meetings and sprints.

Other obscuring factors exist, a team member may not be producing as good code as their peers but they might be contributing greatly to discussions on problem solving or helping fix issues that others have. In this way they could be increasing productivity at the moment but causing problems down the line when others have to fix their code.

Overall this metric is highly flawed if used on its own but with careful analysis of other factors it can be used to pinpoint key facts about developers and how they contribute to the team. These facts may prove very useful for building more successful teams that work together well and have members with skillsets that compliment each other.

Non-Objective Measures

Previously we have been looking at objective methods but subjective methods are also available. Being subjective they can quite easily be biased by views of the person rather than the work they do but so long as they're not the sole point of data they can provide valuable insights. One of the most common measure is peer reviews. Developers should be reviewing each other's code frequently as part of a normal development process. By asking for rankings of the code they have reviewed we can gain an understanding of which developers are producing poorly reviewed code. This might not mean that the code is low quality but it could mean that surrounding documentation is poor or that it's very complex, whether or not that is desired is another question. Ideally, the work would be anonymised to prevent biased rankings based on personality but that may not be possible depending on how work is divided or other circumstances.

Client reviews is also an avenue of measurement, if they feel that the product delivered was satisfactory then that could point to a productive team environment. However this could be very biased and inaccurate! Clients may not realise the flaws that exist in the finished work (bugs or small features that are missing) or may have had entirely unreasonable expectations of the developers.

Meeting with developers and asking them for peer reviews and comments on the process/company are useful tools for determining how the team are feeling about their work environment. The data isn't very trustworthy as people frequently lie and exaggerate but it is a quick filter to point out problems and hone in on issues. These are not perfect measures by any stretch of the imagination but they do make it possible to touch base with people involved in the process and find out the obvious areas where developers may be stumbling.

Where to Compute It?

Online services like Codacy.

These can check for errors in the code, poor quality code, or high complexity. They can't comment on them however, and they might be using metrics that don't apply to your project which give false grades. There is also a security issue of allowing external services access to chunks of your code base and while they may be trustworthy now it is still possible that a malicious actor could use them as a point of access. They are handy in that they automatically update as you push code to a repo and so you can always pop in and see the ranking. They also offer APIs so it is possible to scrape the data from them and see how certain commits change the ranking, for example if certain developers' commits cause problems.

Data from GitHub and other VCS.

These can be used to measure commit frequency and commit size. When combined with various continuous integration tools it is also possible to see if committed work builds or not. For example one of your developers may tend to commit code that doesn't work and then commit fixes afterwards rather than writing properly working code the first time which is a large red flag about their competency if it continues.

Data generated in-house from sprint meetings.

Handed over by the scrummaster or team lead these can be used to measure things such as velocity, cycle time and lead time. Are your teams productive and burning through tickets quickly or are they getting stuck on problems for long periods of times? Do they have issues with implementing solutions or is that they tend to take a very long time to design them? Are they having issues getting to problems in a timely manner and their backlog is steadily building up? All of these sort of questions can be answered by steadily collecting data from every sprint a team has.

State of the end product.

If the clients are happy and feel that they have got good value from their investment then it would suggest that the team as a whole is producing solid work. By determining how much and what each developer contributes to the end product you can see who should be receiving the lion's share of the credit. The uptime of the end product is also an excellent measure. If the service has frequent hiccups or crashes that cause major losses of revenue and user base then the development team has got problems. This also gives rise to other measurements as it is possible to see how they respond to this crisis and how quickly they can fix the mess they have managed to create.

How Should It Be Used?

How to compare them?

So, we've figured out what metrics to use to measure the engineering process in a company and we've picked how we want to keep track of them. Now what? How do we turn all of this data into concrete information that can be used to make a proper, informed decision when managing development?

We need to compare the data, it's no good knowing that one of your engineers spends 17% of their dev time on maintenance unless you have some sort of standard to compare to. Unfortunately, there is no public standard to compare to, you can't just find the average time across all devs, for multiple reasons, mostly that no two companies are the same. A startup will most likely find it's devs spending very little time on maintenance and a lot more time working on key new features while a monolithic finance company will probably have people working almost singly on maintaining legacy code.

From this we can see that several metrics depend on where your company is in it's lifecycle and where it's goals are. If the way the developer is spending their time at work matches what you want your company to be focused on then that's a good start. They're helping to push your company in the direction you want it to be going. Now the questions becomes - how well are they doing it? That's the crux of the issue, once your developer isn't wasting time on work that doesn't contribute to your company's aims, how do you know if they're doing a good job? We have all this data, but how can we know what it means?

One of the most obvious ways is comparing developers against each other to form a relative performance level. This doesn't give any absolute measure of performance but can still tell you who the stronger developers are (assuming you use the correct metrics). We might not be able to tell if Alice is a good developer or not, but we can tell that she's better than Bob so there's that. We'll have to rely on our other metrics of productivity like cycle time and code performance once live. In that way we can see how fast they work and how well the code they contribute to works. Though this is somewhat obscured by it being a team effort there are ways of finding out who is contributing the most to projects by seeing what parts they worked on and how the team feels about their work/efforts.

Doing comparisons against past behaviour is also a very useful way of using our data since it accounts for people being different to one another. By tracking performance over time it is possible to see changes in a developer's behaviour and see if they've been stagnating/declining or improving. A drop in productivity could signal that there's a problem that needs addressing or that it would be good to check in with them and see if something has gone wrong. It won't necessarily tell you how good a developer they are in an absolute sense but it can let you know if they're on the right track to be a better, more productive developer.

What algorithm do you use?

Another simple way of using the data is developing an algorithm that puts different weight on each metric. Maybe you're not particularly concerned with how quickly developers manage to finish projects but you're very particular about the uptime of the finished product - perhaps because downtime could cost a lot of

money. In this way you can see what developers are contributing the most to the goals of the company. Obviously this sort of weighting will need to be revised frequently and used as a guideline (as with so many measurements of software engineering) but it's a valuable tool once properly refined. When combined with code reviews to rate work that has been done and note how problematic it was, that is to say how many issues there were and how severe they were, it will be possible to quickly tell what developers are producing (what is viewed as) good code.

Finding points of concern.

By reviewing all of this data we can pinpoint different areas that are proving difficult during development such as:

Tools that are causing issues

Perhaps your developers are having a hard time managing their open tickets because the software used for it is out of date or clunky and that's what is contributing to the high lead time on fixes. Or else maybe the VCS they use is subpar and causing work to be lost due to poor merges and branch management.

Bottlenecks in your process

If there is an area where teams tend to lag behind or slow down then certain measurements such as cycle time or sprint targets met can show us where the issue might be. For example it's possible that each developer is contributing to the project well but collating and combining their work into a finished product and shipping is taking far longer than it needs to. Knowing this we can start to look at how the process might be changed to make teams work together earlier or merge code more frequently.

Team members that are causing drops in productivity.

Perhaps productivity is dropping because of team based issues - a lack of proper communication or direction - or maybe it is because of personal issues - a developer is finding it hard to deal with the workload. This tends to be mostly looking at correlations rather than showing causes for drops in quality but it serves as yet another indicator of something that needs to be looked at. Many of these measurements work as little red flags that can pop up and let a manager know that there's something to be looked into.

Lots of problems can be solved quite easily if they're noticed early and headed off before they start to spiral - if a developer needs a lower workload shifting some of the work off them quickly can stop a massive backlog building. This doesn't

really help to measure productivity in software engineering but it does provide value to the company that is smart enough to utilise them.

Looking at trends:

Trends allow us to distinguish easily between recurring problems and once off issues. Is a developer a low performer, or is it just that they're having a rough time because of a tough project or personal stress? Are we not dealing with open issues properly or do we just have a large amount coming through from a new service and it'll smooth back out soon? If we can see that something isn't valuable data but is just noise based on the trend then it allows us to filter through the data. Knowing what is useful information means we don't have to panic over every little blip on the radar.

There are several things to keep in mind when using our data however.

Firstly, it is extremely important not to assume what the cause of an issue is but instead to keep probing and researching, compiling as much information as you can.

Correlation does not imply causation! This is a classic mistake to make but when looking at data it is important to note that just because two things seem to be linked does not mean that they are. Just because someone joined a team and productivity dropped doesn't mean that they are at fault, for example someone else could have brought in fish for lunch that week and the smell got on people's nerves or any other number of factors!

Once we have measured we can analyse. It is impossible to analyse properly without some attempt at measurement. Measurement isn't perfect for engineering but it gives us some data to work with and by being very particular about the tools and metric that we use we can still come up with valuable information for our company. There is no one size fits all solution unfortunately, each team and company will have to design their own system for measuring engineering which is a difficult and painstaking process. The rewards are worth it however, with the potential to stop problems early and greatly enhance productivity overall.

Asking for opinions from the factory floor so to speak, is very effective at honing in on the exact source of a problem. People can have a keen eye for what is going on around them and might have insights that higher-ups would never get from their perspective. How people interact can often be hard to see without hearing from those involved, and how your team works with each other is one of the most important parts of understanding their output.

Data should only be used to make decisions once it has been fully analysed. Jumping the gun and rushing to conclusions from partially formed and analysed data will lead to decisions that don't solve problems properly and reduce productivity. Engineers will get frustrated with management that is heavy-handed and seemingly out of touch with the real problems on the floor. Mismeasurement can be just as

dangerous as not measuring things at all, if not moreso from convincing a manager that what they are doing has some sort of logical and factual basis when it does not.

Ethics

What is it ethical to track and measure? Employers have the right to know how much value their employees are adding to the company but engineers are still human beings and they have a right to a reasonable expectation of privacy.

Some metrics are fine - especially publically available ones! Metrics such as number of commits aren't invasive at all and productivity based metrics are similar. Most of the methods of collecting data in the basic and advanced tiers mentioned earlier are perfectly fine. They measure the work that the developer did and how many days they spent on it which is all information that would have been available to managers and employers anyway.

Where it gets murky is when you start monitoring the developer at all times. Is an employer entitled to know how much time an engineer spends working and how much they spend for example taking breaks? What about time spent actively working compared to time spent staring at a screen not making progress? These seem somewhat reasonable, with the second example perhaps pushing the boundaries a little.

Say it was possible to monitor everything they said and what tone they said it in? That would be almost Orwellian levels of surveillance, with every single movement being looked at and potentially being used to judge an engineer with the only thing they had left private being what was in their mind. Most people (hopefully) would consider that an unethical breach of privacy, but then where does one draw the line? That is the question that remains to be settled, how much is an employer entitled to know about their employees and what they do in the work day?

One could also argue that it would be unethical not to use data in your decisions. If you don't inform yourself to the best of your ability when making important decisions about your staff then you are blindly guessing at what is the best path. Employers have a responsibility to make the best decision they can when influencing someone's livelihood, it doesn't have to be the best for them but it should be informed. If you don't take data into account when deciding if someone stays on the team (a decision that could have a massive impact on their life) then I would consider that to be highly unethical as well as obviously idiotic.

Conclusion:

Overall I would say that there are a variety of ways to measure engineers. The problems tend to be that either they are easy to game, or that they don't provide

a complete picture in and of themselves. There is no singular solution that will solve the problem once and for all. Instead it seems that there can only be a stream of homebrewed ad-hoc solutions for each individual company depending on what they feel best suits their needs. It's rather disappointing that there is no magical one-size-fits-all answer but it is also freeing in that companies can focus on doing what is the best for their situation rather than wasting time looking for a unicorn.

Research Sources:

https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0
http://engineering.kapost.com/2015/08/you-can-and-should-measure-software-engineering-performance/
https://stackify.com/measuring-software-development-productivity/
https://dev9.com/blog-posts/2015/1/the-myth-of-developer-productivity
https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams