Measuring Software Engineering

Jerzy Jaskuc, 17341645

28/10/2019

Table of Contents:

Why Measure?

This is the most important question. Why would we use our resources to measure software engineering and productivity of engineers. Is it even possible and cost effective to measure? What can you gain from it? Answer is simple. The more productive software engineering is, the more company earns. However, measuring software is complex, which I will describe later, and therefore requires a lot of resources. But the final result will always show that measuring engineers and their work and then properly adjusting based on measurement results, will net company profits. Not just because company can sieve good software engineers from bad ones. No. It's because company is able to properly prepare and create the environment, where engineers are feeling comfortable to work, can learn from each other and most importantly can see ways of how they can improve themselves. Great engineers make a great company.

Measurable data and how to measure it?

Introduction

First of all we would need to take a look at what data can be measured in the world of Software Engineering. Since it's possible to measure almost everything, the real question here is, what data should be measured and in what way, so that we can get meaningful results out of it.

The other question would be what or who should be measured. Should individual software engineers be measured on one by one cases? Or should the team as a whole be measured? Or should we measure both and include work environment in it as well? Should we measure how well we measure?

After asking these questions we can realize that the scope of measurements can be so big that we would require specific software and specialized algorithms for our measurements. But not every company can afford to spend time on making and maintaining its own measuring software. And that's how companies that are providing platforms for software engineering measurements began. But we will cover it in a separate section.

To answer our questions we need decide what exactly should we measure and that is dependent on what we want to achieve. Do we want to measure productivity of an engineer, productivity of the whole team or maybe effect of working environment on the engineers[i]. They are all intertwined, but we should pay attention to the different

details, depending on our intentions. Of course, these are just a couple of examples, but let's take a closer look into those.

Measuring engineers productivity

First thing that comes to mind would be measuring what software engineer produces.

That is:

- Lines of code written
- Number of commits
- Number of keystrokes compared to lines of code written[ii]
- How fast the task is completed

In reality, we can roughly combine those into one thing. How fast is the engineer at writing code?

Now to expand on the points. Measuring lines of code alone is not a good idea. Why? Because that doesn't mean that the code is good. Quantity over quality is rarely a good thing in any industry, and Software Engineering is not an exception here. Anyone can a write a lot of code to do a small thing. But not everyone can write a couple lines of optimized, efficient and readable code. Measuring amount of lines of code on their own would encourage inefficiency. But combined with something else, it can make sense to measure lines of code. We can measure how many lines of code per task engineer has written. But instead of many lines being a good metric, we can say that having less lines is better. This would be slightly better way to measure, but it can be detrimental to the readability of the code. So we would need to add readability in measurement. But will discuss it a bit later.

Measuring number of commits is more effective, but an important rule should be brought up. Commits would have to be meaningful changes that work. If commit is not fully working (i.e. work in progress) a person could just commit very small changes to boost the numbers. That would make our measurements ineffective. The other problem with commits is the size of them. We can't really just take number of commits into consideration. Different problems require different solutions. Some problems are bigger and might require a thousand lines of code, while some small bug fix would require 10 lines. Now imagine situation where one software engineer is fixing bugs, while another is rewriting the service to use the new API. The work they put in might be the same, but number of commits will differ.

Therefore we would need to take size of commits into consideration as well. Of course, maybe company wants to enforce smaller commits, so that reviewers can get to them faster and also spot more mistakes, which would make overall code more polished. Smaller changes are also easier to revert in case something goes wrong. All depends on intention here.

Some sources suggest measuring amount of keystrokes it takes for a person to write a certain amount of code. In my opinion, number of keystrokes is the worst metric imaginable, that you can measure. Every person have its own preference in terms of how he wants to code and the more comfortable it is for him, the more productive he will be. If a person will need to learn a lot of shortcuts that he doesn't really want to use, just to cut down on keystrokes and have better metrics, that's just a waste of employees working time and money loss to a company.

Measuring how fast tasks are completed is good enough metric, but it's also not without caveats. Speed is not everything, task has to be completed with reliable and solid solution. There's always comes a maintenance cost with a software solution. If solution is rushed out in the first place, the maintenance cost will grow quickly, which is not the goal of this. This metric will have to be combined with all above, to be meaningful (with exception of keystrokes), Otherwise it will be abused, like any other metric on its own.

This were just the things you can measure overall. But software engineering these days have a lot more to it. Metrics above can be abused without extra characteristics that we should measure as well.

These are :

- Lines of tests per lines of code
- Test coverage
- Code reliability

All of these could be combined in one again - how well tested and solid is the piece of software. But once again, nothing is that simple. Not only we need to combine all these metrics, but also we would need to combine them to the ones we preciously mentioned. Why? Answer is simple. We can't test reliability of a piece of software and it's test coverage without actual piece of software. But we also can't measure how well did engineer write his code without those reliability checks.

Code have to be tested. At the current industry state, testing in production is almost never an option. The competition is big and there's a very small room for any mistakes. Therefore we need tests. Lines of tests per lines of code can be a good metric, but is has to go together with test coverage. It's always possible to write useless tests that will test the same thing, but that's not really the point of testing, is

it. The important part is to catch the edge cases that can cause software to fail. In that sense, by test coverage here, I mean coverage of all possible edge cases that this piece of software can encounter.

Other important part is code reliability. Since it's quite broad term, in here I would describe it as how dependent is your code on the code of others, beyond of your or your teams control and how well does it scale. If a software engineer writes code, that is minimally dependent on external 3rd party libraries and which also scales well (doesn't break under heavy load, have load protection mechanisms) and requires minimal maintenance, I would hire that person any time. To be reliable, it needs to be well tested in a first place, so this metric have to be placed together with testing.

That's all that we can realistically measure, when determining engineers productivity. As you can see, we cannot use just a couple of metrics, as they can be abused or be inaccurate on their own. But combined, I believe they can provide a very good picture of how productive and how good is given Software Engineer.

Some people would also consider measuring how happy is the person and how satisfied he is with the job[iii]. That's an interesting way of measuring and I would say it could work out well. If a software engineer is happy to come to the job and work that usually means he has a good grasp of what he is doing, has things under control and probably has a bit of autonomy, so that he can decide what he is doing. It also means that a person doesn't have any distractions, as they are tend to be annoying, especially when you are trying to work. Being annoyed usually make a person rather unhappy than happy. In this case happy = productive, right?

Problem arises when we want to measure job satisfaction and happiness. We would need to concern ourselves with proper metrics for that as well, which would require some research. But once done, it can be a very valid measurement of productivity and if combined with traditional ones described previously, it can help fully measure persons productivity and skill. Therefore, measuring work environment is really important as well.

Engineer in a team productivity measurement

We also need to remember that in todays world, no serious software engineer works alone. It always involves work in team. Even the best programmer, in reality, won't be very productive, if he can't deliver his work to the team. Therefore we need extra metrics to measure, in addition to measuring code itself and its tests.

These would be:

- Code readability
- Documentation of the code

Both of them should be measured together, since documentation of the code makes it more readable as well.

Why is it important? Because all the software has to be maintained. It's rarely the case, where you write your piece and neither you or anyone else will see it again. People will rewrite parts of your code, build on it or use it as a dependency. Or your team members will look at it to see if they can use it or to see if the feature is done yet or still have to be written. If code is unreadable and not understandable, then in the worst case, you can say it wasn't written at all and the time was wasted. If it's completely unreadable and usage is not undocumented, then it just can't be used and it will be deleted and rewritten by someone else. The more readable is the code, the more chances that it won't be pointlessly rewritten, that it will be used properly, and other people will spend less time on trying to understand it. That doesn't increase persons productivity directly, but it nets overall gain in productivity, on account of that person. That's why I think readability and documentation is also very important to measure.

In terms of non-written metrics, it's also should be measured how well person is able to communicate his ideas and thoughts to the team. But I won't describe it here, as it's going more into management field, in my opinion.

Conclusion

As it turns out, measuring software engineering is no easy task. However, it's also not impossible as some people would imagine. To get a proper full picture of productivity, all the metrics and factors described previously, have to be included. Without them all, metrics are either incomplete, inaccurate or can be forged. And in my opinion, there's no worse thing than being measured incorrectly. But more about it in ethics section.

Algorithmic approaches

Having metrics that we need to measure is nice but how would we use the results of the measurements? How would we decide whenever given result is good or not good enough?

The most basic way would be to gather performance metrics of everyone, derive an average and see who is doing well by being above average and who is doing not so great, below the average. Problem with this way is that it will create a toxic working culture, in which a person, instead of improving himself/herself might as well

undermine someone else metrics, so that average stays lower. This can even lead to so called crab mentality[iv]. So that's a no go.

It's possible to expand the scope to the whole industry, check how competitors teams do. This is possible with for e.g. GitHub API, which is widely available. When the only way to stay above average is to improve, it does not create a toxic environment and personal improvement is incentivized over ruining others performance. It's still not a perfect solution, as every company is different and you can't always compare them.

To improve that we would need to consider an impact of a person on the company, both direct and indirect[v]. That would include all metrics we mentioned and compare it to how much profit company gained from that. It can be measured either in revenue, or saved hours of work in case of work on internal tools. It's also possible to measure how well work was aligned with companies priorities. That should give a good background for comparison when using results of measurements to decide what to do.

Of course those just a couple of examples. Real tools use much more metrics and it's pretty common to use Deep Learning to establish patterns and predict possible outcomes or suggest ways of improvement. With amount of data available, it's possible to train neural networks to do the job, and give pretty accurate results.

Platforms available

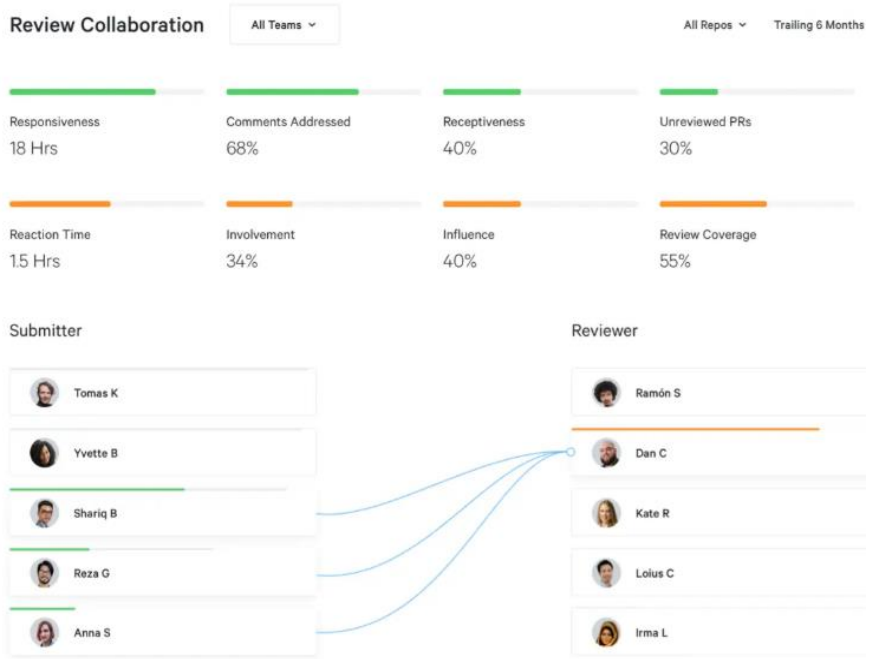
Since, as we established, measuring software engineering is important, but also resource consuming, a lot of platforms were created to provide measurement software to their clients.

Let's discuss a couple of options.

There are some ready made products, like GitPrime[vi], that offers large variety of available tools and analyzers. It can analyze personal performance, team impact, team growth and similar. Moreover, based on previous measurements, it tries to predict outcomes of scaling efforts and suggests sprints plans to the teams.

What's most interesting however, it can even measure how collaborative are people within and between teams.

Here’s an example from their website:



This tool is also used by relatively large companies like LogMeIn, VMWare or PayPal and they claim to raise efficiency of software engineering by about 25% (up to even 85% in specific fields).

Considering companies earn billions in revenue, 25% increase is massive, so this goes to show how important are software engineering measurements, especially at a large scale. Software like GitPrime can allow better scaling of the company in terms of team management.

This however, requires a lot of data and despite privacy agreements, some companies might not want to disclose their statistics to the third parties. But for that case, there’s a lot of API’s available for use, even from platform like GitHub. It’s possible to use their REST API to get any information regarding specific commits, people, teams or even trends throughout the whole platform. Therefore, it’s possible to use it and make your own software for measuring productivity.

In the last case, if company uses it’s own version control tools and such, then they probably also have their own measuring tools, specified for their needs. This is the case with the largest companies like Google, Microsoft or Facebook. Measuring Software Engineering is important and so these tools must exist, but how exactly these tools look, I can’t say, since it’s a company secret.

Ethics and concerns

Now, knowing all the metrics that can be measured and all the platforms that provide us with tools to measure, let's consider ethical aspect of this. Is it ethical to measure productivity of software engineer? As we saw, this requires a lot of metrics, therefore we would be gathering a lot of information about engineers. Not only it raises privacy issues, but also imposes a question. Can you really measure a person? Everyone is different and we are not robots. We all make mistakes from time to time, therefore someone might be doing well in multiple quarters and then did bad in the other. Would it be enough of a reason to fire that person? What if the person is not doing that great in coding, but is essential to the team spirit and is helpful to the others? Then his overall productivity might be better on different metrics. All these questions suggest a couple of things. Measurements should not be strict. Every person is different and will be of value in a different way than the other person might be. Measurement should be taken in a prolonged period of time and average should be taken. Everyone has good and bad days and basing decisions off one measurement is far from reliable. And most and foremost its also important to keep in mind that measurements don't show everything about a person (unless your measurements are so big that they can be considered spying), so they should be taken with a pinch of salt.

Since measuring raises privacy concern, can we just not do it? Unfortunately, in the end it's all about business and money. And in order to produce more effectively, you need to keep raising the productivity of your workers and keep the most productive ones. Without a solid way of measuring software engineering, that would be impossible and would rather leave it to intuition, which is not the most reliable way to assess productivity.

All in all in my opinion being measured is not a bad thing, as long as you are measured accurately. Being measured can help you improve yourself, by showing you areas that need improvements. It also gives an incentive to do so, since you can see the results of your work.

That's why it's also important to use proper metrics, since in my opinion, there's nothing worse than being measured incorrectly.

References:

[i] https://nortal.com/blog/the-myth-of-developer-productivity/

[ii] https://www.mysammy.com/how-to-measure-productivity

[iii] https://redfin.engineering/measure-job-satisfaction-instead-of-software-engineering-productivity-418779ce3451

[iv] https://en.wikipedia.org/wiki/Crab_mentality

[v] http://www.cs.umd.edu/~mvz/cmsc435-s09/pdf/slides16.pdf

[vi] https://www.gitprime.com/