Measuring Software Engineering Performance and Productivity

Report by Matthew henry

The accurate and consistent measuring of Software engineering performance and productivity is an ongoing challenge in the world of computer science. For both employees and employers in large and small scale operations, the ability to, with clarity, measure the performance, productivity and ability of engineers in a variety of disciplines is paramount in improving workflows, ensuring consistent development momentum and reducing technical debt.

As with a lot of aspects of the field, the problem though simple in description, has proven to be very difficult to solve in practice. No approach has become a “silver bullet” in solving the issue, but many different and increasingly interesting approaches have been broached with the hopes of providing an insight into the operational performance of software engineering as a discipline, with some unfortunately being utterly ineffective. Due to the difficulty in overcoming this problem in the past, it’s often been slated as *impossible* to measure software engineering productivity, and that it just **can’t** be measured properly.

Personally I see this as perhaps an over-exaggeration, just because there is yet to be a one-size-fits-all solution does not warrant the thought that there is no solution at all. These difficulties should be seen as a challenge to refine, rethink and ultimately reimagine how we think about software engineering and how we can come to measure it in the future, both computationally and heuristically.

Metrics and measuring

The first question that comes to mind is what exactly are we measuring when we say performance or productivity and that question, by in large, is the root of the difficulties in actually creating an accurate and consistent model for software engineering as a discipline. There are arguably some obvious metrics that could be used but, unfortunately, there are also a number of pitfalls associated with the majority of them.

For example, an often thrown about solution is measuring lines of code over time. At first glance this can seem very sensible. A large scale project, at its root, needs a large code base to support it, and so if an engineer is writing, say 100 lines of code in an hour vs another engineer thats writing 50 an hour, then the first engineer must be more productive, right?

```
1 #!/usr/bin/python
2
3 print "Hello, World!";
4
```

"Hello, World!" program in Python

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("Hello, World! \n");
6    return 0;
7 }
8
```

"Hello, World!" program in C

Figure 1. Python code vs C code

Using the figure above as an example however, we can see that this metric might already be completely flawed. If the two engineers are using different languages, systems or even tools, the number of variables involved outside of just “Code per hour” begins to completely skew this metric. Even when using the same languages and by in large the same tool kit, depending on the area, complexity and also style of coding, the idea of using such a single rigid metric becomes progressively more unpalatable. The question becomes even more broader when refactoring takes place, how does *removing* lines of code factor into a measure of productivity or performance.

Perhaps realising this, you decide to pick a more general metric, one more agnostic to the code itself. With something like a source management system like Git or SVN, commits, branches, repos and more are all now recordable metrics. Taking commits as an example of something to measure leads to a similar issue. As conflicts in commit style, reasons, additions, deletions and reviews all come into question. You could be the most “productive” member of the team with the most commits, but if you’re only committing one line change at a time is that not just slowing you down? A

clever developer is going to skew these metrics while not rot really giving a whole lot back to the team

Maybe you'll take another step, an abstraction from the tools and instead focus on bugs fixed or issues resolved, 'Everyone knows that debugging is twice as hard as writing a program in the first place'[2], so if you solve a bug you must be twice as effective a developer. This on the face seems even more sensible than the previous metrics, however once again upon looking a little more deeply into it we find similar flaws. In as much as we would wish it were the case, not all bugs are created equal and neither are solutions to them. In a rushed environment a fix may even introduce more bugs, which if we were to take this metric at face value would in fact be seen as *the most productive thing you could do!* At an even more ridiculous extreme, should the code be bug free, then your developers would be considered utterly unproductive as they'd have no bugs at all to fix. Once again such a singular metric proves not only ineffective, but potentially detrimental to the project.

The next logical iteration with the idea of metrics, comes from a marrying of a number of potential metrics, perhaps creating a more heuristic approach to measuring productivity as opposed to relying solely on an empirical measurement. An example of this could be Technical debt. This comes about when a project has to abandon, delay or otherwise skim over a certain feature or service, with the hopes of coming back to it later, hence the term "debt". Measuring something like the inverse of the amount of current technical debt in a project could make for a good productivity metric. However we run into the issue where by we are measuring

performance on the *lack* of a particular metric, which is a flimsy idea at the best of times, and perhaps not the solution. Certainly an improvement however, and in industry is often used to see how well a sprint of project went in regards to time and resource management.

Another common blended metric in the field of software engineering is ‘Churn’[3]. Development Churn is a term that refers, generally, to the amount of times or time spent rewriting old or already in production code due to changes in deadlines, outcomes or specifications during the development cycle. This is most often attributed to Agile development as it’s not uncommon for customers to change their mind in ways that hinder the development process. This leads to an inherent cost associated with task or context switching as a developer is ripped out of their flow and “churned” up in the management cycle. At a high level, this is often described as the number of lines written minus the number of lines changed on a particular project/sprint or development milestone. Although not perfect by any means, this homogenization of code metrics and more heuristic observations is a very commonly used as a means to refine processes. It does however lack the more fine tuned metrics or analytics that most developers and managers would like. After all, numbers don’t lie.

In a similar vein, something like customer satisfaction or developer satisfaction finds a similar pitfall. Although it’s easy to say that if the developer and customers are happy, then the developers must be productive and efficient enough to keep that up.

Although that makes sense, it doesn't provide any real tangible measurements that could be given to reflect that.

As we can see there are a lot of possible metrics and means to measure Software Engineering performance by, there really is no perfect solution that's been devised as of yet. It's easy to see why it might be considered impossible or untenable to measure, but thankfully that hasn't stopped people trying. A whole family of algorithms, tools and systems exist and are constantly being added to with the hopes of not only measuring Software Engineering but also with the intention of improving efficiency and performance. Although it's often stated that "When a measure becomes a target, it is no longer a good measure"[4] [Goodheart Law], I would argue that no measure at all is much worse.

The underlying issue really may be the discipline itself. As Software engineers we strive for measurable results, measurable performance and measurable improvements in everything we build and design. The frustration is, that perhaps **we** might not be able to be treated in the same way, despite the numerous attempts and experiments done to strive against the fact.

Platforms, tools and computational solutions

Despite the numerous difficulties found in nailing down a metric, in devising an appropriate algorithm and most of all implementing it fairly and ethically a family of

tools have been devised, developed and produced in the hopes of providing a level of measurement and transparency in the software engineering world.

These tools range from all encompassing source management and ticketing tools like Microsoft TFS to barebones issue boards like Trello, or GitHub projects. No matter the development paradigm, be it Kanban, Waterfall or Agile, a platform or tool exists that attempts to provide engineers and managers with the tools and metrics to efficiently handle the challenges involved in software engineering endeavour. No matter the system however, it is important to weigh up the metrics it measures, how it measures them, the analytics it provides and most importantly, the value it provides to the team or group should it be adopted by you. Even the most sophisticated system can be all but useless if the context its applied to does not suit it.

Before ever implementing programmatic solutions for measuring software engineering, some attempts were made to codify and define “software quality”, for some this became the capacity for software to meet requirements[5] while others it was a more ethereal measure of customer satisfaction. The CISQ defined 5 characteristics that would be used to measure the business value of a software engineering solution.

- Reliability
- Efficiency
- Security
- Maintainability

- Size

However, unfortunately at the engineer level, these metrics provide little to no actual insight to how performant an individual engineer is on a project by project basis.

The first family of tools born to record software engineers was almost solely manual. A clunky system detailed by Watts Humphys book on the Software Process[6]. This involved manual input of a huge number of data points on a large volume of forms. Despite being very thorough, it was also very intrusive which, after two years, had an obvious negative impact on the performance of the engineers involved. Obviously this raised the question against manual records as a whole. If your engineers spend as much or more time attempting to record their performance, is that not a waste of potential performance to begin with?

In order to move away from the obvious flaws of manual recording, and with the rise of test driven development, Zorro was developed. Zorro came to life as an automated system designed to determine whether a developer is complying with test-driven development practices, with the hopes of improving the overall performance of a software engineering team following TDD practices. Zorro and tools inspired by it are still used today as hooks in IDEs like eclipse.

A blend of automated and manual systems came in the form of tools like Jira, created by atlassian. Jira is an issue and project tracking system based on the agile methodology of software development, similar to Microsoft TFS and Github projects. Jira provides tools and hooks for planning, tracking and understanding software

projects and the engineers involved in them. It's a very popular tool due to its huge amount of integrations with things like CI pipelines and test toolkits. Jira and tools like it require a certain amount of manual input in order to keep track of tasks and issues, but provided it's kept up to date with relevant information, it does provide a pretty comprehensive overview of the entire development process and a general vision of the productivity and value of an individual engineer on a team. It should be noted however that in the vast majority of cases, these planning and tracking products lack the fine grained analytics that we would hope to see when actually empirically measuring software engineering performance.

In recent years however some more worrying tools have been created, less directed towards the work done by an engineer and more focused on an engineer themselves. Companies and groups see that if an engineer is satisfied and happy at work, they're more likely to be productive and provide value to the business. In order to track this, measureable "happiness", and with the rise of wearable technology, many companies are using tools like fitbits and the steel series. By providing these technologies as "perks" its easy to gather information on individual engineers. Even in my own workplace, its common to share your heart rate during a particularly stressful or difficult production change. Metrics like these are only the beginning, other studies attempt to quantify how a team or an individual communicates[7], studies done by companies like Humanyze attempt to show that even this kind of metric is measurable[8]. Using Smart badges to track tone of voice, conversation topics, location in the office and a number of other factors, producing upwards of 4GB of data per day per employee. This approach follows the "A happy engineer is a

productive engineer” idea for metrics gathering, but must elicit some fear that whatever algorithm might happen to be tracking you, might not think you’re productive or happy and then lead to reprimanding utterly outside of your control.

As data tracking, collection and mining software and tools become more prevalent and more invasive, now more than ever is the time to question how ethically sound they are. Imagine starting a new position as an engineer, given your new desk and brand new fitness watch, only later to find out that your chair is recording your posture and how long you sit at your desk, your watch is recording and storing your health and even your office itself is keeping track of how often you leave your desk to chat with a coworker. Each and every one of these things is tracked for a reason and could produce a point for or against you within the business, and unfortunately, as we can see in places like China, systems like these can be extremely unfair and dangerous. Imagine having your pay deducted automatically because your chair has detected you’ve spent under the expected number of hours at your desk in a given week.

Realities like these really aren’t that far away, and in some cases are just as ethically dubious as they sound. Although the search for measurability is a good one, without the correct restraints or oversight it can absolutely lead to more oppressive outcomes than good ones.

Algorithmic Approaches

Along with the tools to complement them a large number of algorithms and approaches have been created to both improve engineering productivity as well as to make measuring it as easy as possible. The most traditional of these methods is waterfall[], which follows a more monolithic approach to development. The methodology is at its heart a relatively simple, following a logical set of steps

- Analysis
- Design
- Code
- Test
- Release/Maintain

One development cycle will encompass all the above steps and result in a finished product, theoretically at least. However as time has gone on an issue has been found with this system, the change request. It's extremely common for a customer or group to modify the requirements/spec given to a software development team as their needs change, this often leads to huge technical debt and churn in a project governed by the waterfall method. Although it logically would make the measurement process easiest, as a function of the time taken to get from Analysis to Release, in practice it fails to encompass the changing nature of development in the real world.

To combat this, the Agile method was born. Similar in some ways to Waterfall, agile instead focuses on a number of smaller sprints of a similar style of cycle. Each focused on a particular part of the overall and planned out ahead of time. These sprint cycles typically last a set number of weeks. This allows for a customer to have an input at the end of each sprint, thus making room for the changes that are common place in the software development world. Agile however comes with a caveat, it requires far more managerial overhead to get right, as churn and technical debt can and do become rife if changes come too quickly for a team to handle or if time scales are not correctly mapped out. Agile tends to work best when a team can measure and react to their own available pool of productivity, not so much the other way around.

Another slightly more esoteric approach is KanBan, this approach involves splitting up a task or project into a huge web of issues or subtasks. These are then put on a KanBan board with differing weights and the team then picks which tasks to work on at a given time. With regards to management this method is likely the most hands off of them all, providing a simple metric with which to measure productivity. Tasks done multiplied by weights, divided by the time taken. However, this system is easily abused by less enthusiastic engineers who take on a large number of very small tasks instead of core features. It’s not hard to imagine that the project could fall quite far behind its actual Key goals while also seeming to have a high internal velocity.

As we can see, just like there are plenty of tools and metrics for measuring software engineering, there are a large number of processes, algorithms and methods

attempting to provide and use those too. We have yet to come down to a silver bullet, but provided the context is right and the team is willing, these processes can be very effective.

Conclusion

To sum up this report, as an industry and discipline software engineering has a plethora of tools, metrics and methods for measuring and improving productivity and efficiency. Despite the cries that software engineering is akin to an art, with our attempts at measuring being only hindering the flow of engineers we can quite obviously see both the benefits of the drive to do so. Without looking for a solution, the tools, tricks and methods we as an industry use to achieve the monumental tasks that exist for us I highly doubt technology would be where it is today.

Although I have my apprehensions at the ethical concerns that the new age of data is bringing to light about how we gather, use and abuse our data and that of others I think that by in large we are stepping in the right direction. We may never find a silver bullet or holy-grail, but if the evidence is anything to go by, we’re only going to get closer by trying.

References:

[1]http://www.circuitbasics.com/how-to-write-and-run-a-python-program-on-the-raspberry-pi/

[2]Brian Kernighan, "The Elements of Programming Style", 2nd edition, chapter 2

[3]https://scottbarstow.com/churn-in-software-development/

[4]Goodhart, Charles (1981). "Problems of Monetary Management: The U.K. Experience". In Courakis, Anthony S. (ed.). Inflation, Depression, and Economic Policy in the West. pp. 111–146. ISBN 978-0-389-20144-1

[5]International Organization for Standardization, "ISO/IEC 9001: Quality management systems -- Requirements," 1999.

[6]1989. *Managing the Software Process*. Addison-Wesley, Reading, MA.

[7]https://blog.usenotion.com/5-essential-software-development-metrics-to-measure-team-health-a08f089528d4

[8]Chen, Hong-En, and Miller, Scarlett R. "Can Wearable Sensors Be Used to Capture Engineering Design Team Interactions?: An Investigation Into the Reliability of Sociometric Badges." Proceedings of the ASME 2017 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 7: 29th International Conference on Design Theory and Methodology. Cleveland, Ohio, USA. August 6–9, 2017. V007T06A024. ASME. https://doi.org/10.1115/DETC2017-68183

[9]https://airbrake.io/blog/sdlc/waterfall-model