Measuring Software Engineering

Owen Gallagher

19 December 2021

Contents

1 How to measure activity **2**

1.1 Introduction 2

1.2 Task boards 3

1.3 Bounty-hunting and testing 4

2 Platforms to use **6**

2.1 Task management 6

2.2 Outcomes and data analysis 6

2.3 Setup for analysis 8

3 Computations **9**

3.1 Evaluating bounties 9

3.2 Counting lines 9

3.3 Positive engagement 10

3.4 Rating friendliness 11

3.5 Finding laziness 12

4 Ethics **13**

4.1 Scrutiny 13

4.2 Intrusiveness 13

Chapter 1

How to measure activity

1.1 Introduction

To measure engineering activity properly, I think it is required that software engineering is strictly defined as to better understand what needs to be measured. It is evident that software engineers are also programmers, though the term programmer of course might be insufficient. While a software engineer will go into a new job well-versed in programming, the future of their career could be said to rely on various other factors. A software engineer applies mathematical analysis and the principles of computer science to design and develop software. It would not be a stretch to say that these things are par for the course because they must efficiently manage and extend a code base.

Thinking about the practice of software engineering illuminates then what I think is at the core of the development process: organization and cooperation in a team. Just like any other engineering-based discipline, it is immediately apparent that some difficulties arise with the mindset a person can fix any problem with foreknowledge alone. Positive aspects on the job should include asking the right questions. A bonus on top of this would be the tendency to write and keep notes. A software engineer should ask questions to understand their notes. Code is complex, it should be expected when a person comes into a job and does not know why things are done a certain way that he or she asks around or scours documentation. At the end of the day we can take as a given such things are blatant enough that they need not be included in most job descriptions but sometimes they still are because of how important those qualities always will be. To over-engineer and not communicate would not only make it difficult to manage a code base but will not make use of the foundation laid by past participants in the engineering process.

So I think important metrics for software engineering activity are engagement as well as code quality. However if we were to examine how well they can describe a person's aptitude or engagement something becomes very clear. Take for example someone writes concise code that integrates with a colleague's program. They have demonstrated an understanding of context and solved the problem laid before them. But to look at this example in isolation could be detrimental to the accuracy of such an assessment. The mistake here could be in my opinion that it was not innovative or practical. Was the addition of this code warranted, regardless of whether it was well written and friendly? My point is that careful use of one's own time as a software engineer is a healthy trait of an engineer and easily forgotten in lieu of instant gratification.

> When I interview people I tell them, 'You can work long, hard, or smart, but at Amazon.com you can't choose two out of three.'

CEO Jeff Bezos had this to say [1] about working long, hard and smart at one of the most valuable companies in the world Amazon. It is peculiar to me because he presents an option followed by a very interesting punchline: that we cannot choose to be simply clever and hardworking but also pour our time and resources into our work. The basic premise of what Jeff Bezos shares about his methods is nothing special but it often leaves little room for interpretation. It is quite clear that there is some standard for people working in all fields to for example not just clock out after five on workdays.

I mention this because not only is time of great importance to the people hiring software engineers, but because there is a great misunderstanding with regards to exactly how to use it. To work for an hour's pay is to work so that when you are not working you have done all the work you needed to within that allotted time. Although there will be exceptions, time is money because what transpires over any given amount of time overall affects the product. Moving away from more general terms it is presumed a truly good software engineer on paper works hard and smart so that they need not put in long grueling hours afterwards! I acknowledge that might be a divisive idea for anyone and especially for programmers, what with debugging and all being laborious and stressful. I admit to sometimes working later than I ought to.

I conclude then that to adequately assess software engineering a metric can be used that looks at merit, engagement and code quality. It is immediately obvious that all three characterize individual commits to a repository of code. Commits will often consist of code changes, comments and be related to the rest of the repository via branches. I think by scrutinizing code, legitimacy of commits and progress made anyone can assemble a profile of software engineering. I propose this level of introspection can be achieved by means of branching out from the code layer of a project and into the management layer.

1.2 Task boards

Project management nowadays is more popular than ever. The basic role of project management is to capture success and prevent failure. These methods reassure firms and offer some certainty. As The Tech Report states [2], Trello is a great tool to use; Trello is a task board app for project management and coincides with GitHub in its mission to encapsulate development cycles which can clarify the focus of any particular sprint or period of time for engineers. It goes without saying that this presents something indicative of the current state of things: that there is an overwhelming amount of context about what software engineers have done and will be doing. Building associations with the code base of a project allows us to better understand what went into writing it and essentially to understand the engineering that is behind the product and not always shining through. The adoption of these technologies at a rapid pace will change the landscape for grasping new techniques employed by developers when programming, things unknown to even those developers as they write code to complete their tasks set for them, hopefully with better regard for why they are essential to a project and what they could contribute in the future.

Agile development and its popularity has a major role in my vision for measuring activity because it involves a similar engagement with the philosophies I have espoused thus far. Agile is a methodology or mindset which consists of several iterative and incremental software development methodologies. It is not always possible to gather all the requirement and clarifications at the time of starting the development. So the Agile Model is specifically designed to satisfy a customer with continuous product delivery. By frequently developing a working product there are constant adjustments to the allocation of resources and control to steer the entire development process.

In the same way business have grown to adopt incremental project management systems that do not stifle creativity and cleverness a more modern way of measuring software engineering can come out of incorporating the management process in auditing or governing software development. In terms of commits, how does a new software engineer break down a problem? Does quality of code suffer because of the lack of focus attached to an experimental feature? Are people making frequent and large changes to a repository they have no history working in? These are questions I mean to ask and form the foundation of what I think is the most current way of reflecting on software engineering.

One of the principles behind the Agile Manifesto is as follows: Working software is the primary measure of progress. [3] Simple though it may be, the implication is undervalued. To look at the many lines and lines of code and the history of work by the users on the versioning system of choice is all well and good. But when we examine the results and what came of the code we can attach a metric of ”meaningfulness” to code.

1.3 Bounty-hunting and testing

Martin Fowler, who discusses agile methods [4] also discusses extreme programming for its focus on test driven development and how it stood out for simply that. Helping to realize agile methods as real actions, it was an innovation that allowed for the incremental success of development cycles everywhere. We can attribute this achievement to careful examination of software engineering by people. By taking inspiration from it we learn that the distinct point the extreme programming system advocates is that code should achieve something and not only achieve something but actually regardless of code just complete a task or pass a test. So why not first and foremost to effectively find out how essential a person's work is, we reward relevant work done, relevant tests passed? This would greatly align with how people communicate about projects as opposed to repeat the mistakes of old such as the counting of lines written.

By way of "bounty-hunting" there could be an allocation of tasks with their significance explicitly stated. Attention is given to projects that require attention, merit is based on the balance of important tasks and the creation of important tasks. By attaching a developer to various roles an informed and unbiased understanding of how they get results can be laid bare without unnecessary focus on how desired results are achieved. I think in this respect coding is more of a secondary role and merely an engagement indicator more than a progress indicator. With bounties there is motivation to do work and a better communication of what work a person can do with feedback or rewards in mind. Simply seeing what a person pursues when encouraged tells of what will motivate them in future. This in itself can gauge the latent potential of a software engineer and streamline a feedback loop for software development.

In conclusion, to measure software engineering I suggest the aspiring analyst to set goals for developers and interpret incrementally the responses positive or negative to assigned tasks. Via code quality and engagement to different tasks bounties can form more easily over time and provide opportunities to find the full extent of a person's capabilities. Furthermore with the right execution a metric can be made of the impact tasks have and whether people have tasks with impact will speak to a developer's achievements. Code quality will serve to show efficiency if tasks are compared in terms of how elaborate they might be and as a result less code upon completing a task will be indicative of aptitude. But not to only reward potentially finicky solutions, engagement in foreign places in the code base to them will also be accounted for. This means we have an equation of adaptability to productivity, two parts that cancel the other's flaws.

To provide some anecdotes for better conception of this idea, consider two people developing an update for a feature. One developer has been on the job for a while now and has the feature as their new bounty. The developer alongside them has already worked elsewhere and is now on a trial run of this feature, which they are not familiar with. Perhaps the new developer will write many more lines of code for the feature to make a good impression. For the sake of argument it is many more than they usually would write on average in a single development cycle. But there is no change to the number of tasks completed compared to when this developer was not present. It does of course imply great engagement for that developer. Lines removed and added in my opinion would be one of the most novel concepts for understanding when a developer deems it necessary to remove lines. This in turn allows for a back and forth between developers to be illustrated in the code base itself. If the veteran had a history for deleting code they did not write but had not deleted in this instance I would consider this to be a good match up as well as a great statistic on the new developer's performance for a category of tasks. That would be what I expect from measuring their activity.

Chapter 2

Platforms to use

2.1 Task management

To atomize the information I set out to look for is no simple task but the resources from which to take are readily available these days. I mentioned Trello offhandedly as one of the biggest components in project management and by extension software development. Platforms like GitHub also provide very similar services which allow users to identify problems and fix them on a schedule if that suits. Whatever an organization uses both provide an API for the information that interests me. [5] [6]

Linking user accounts for commits and for task boards would be a simple task and allow for the type of oversight that leans in on the project management side of group programming. From there it can be made out what tasks are being done and which commits are done in this stretch of time. This already will create frequency data for users on the commits level.

Besides the access to repository data there is also the external data from websites like GitHub that do not necessarily pertain the code in a repository. I think this is interesting as a source for developing engagement algorithms of some kind and manufacturing a process that looks at all angles to not just fault developers for a single aspect of their entire approach.

2.2 Outcomes and data analysis

This Gartner survey [7] provides insight for me about productivity metrics. Just about more than half of leaders are rapidly scaling their digital businesses based on the idea that tracked metrics ought to be business growth from their choices. Productivity metrics need to be verifiable available and repeatable. Following on this project planning needs to be evaluated for its cost, effectiveness and viability going forward. The takeaway here is that we need specialized concepts when dealing with data in good faith. We can start by going over some common ideas in agile scrums and seeing how they are currently analyzed today. For context, story points are units of measure for expressing an estimate of the overall effort required to fully implement a product backlog item or any other piece of work.

- A sprint burndown report communicates the complexion of work throughout the sprint based on story points. The goal of the team is to consistently deliver all work, according to the forecast.
- Team Velocity metric accounts for the "amount" of software your team completes during a sprint. It can be measured in story points or hours, and you can use this metric for estimation and planning.
- Throughput indicates the total value-added work output by the team. It is typically represented by the units of work (tickets) the team has completed within a set period of time.
- Cycle Time stands for the total time that elapses from the moment when the work is started on an item (e.g., ticket, bug, task) until its completion.

A great example of these different viewpoints of activity being acknowledged could be seen in the product demonstrations for the apps WayDev and NDepend. This brings me to a popular discussion in this sphere, about Lines Of Code (LOC) which is the counting of useful lines of code, usually statement lines or operations as opposed to brackets and such. In this article [8] we get an informed counter argument about the legitimacy of this metric. It is important to pay attention to the fact that despite all our attention to other avenues they could all lead back to this arbitrary counting of lines, which in the end is a very attainable thing for software that sets out to simply measure performance. Drawing from well established author Fred P. Brooks Jr. and his book The Mythical Man Month [9] there is in my opinion a lot of inspiration that is relevant to my own discussion that follows on these data analysis ideas at play in this section.

The article by NDepend [8] discusses how in the long term LOC inevitably rises back from any previous fallings back. This makes sense because starting from nothing you cannot really go backwards in terms of lines. As a project gets larger it becomes increasingly difficult obviously to keep it small. It is without a doubt there should be a steady increase regardless of habits in general. What does this mean? It can be used as an estimation tool and rate monitor as opposed to a strict day to day goal. By getting an average based on accrued LOC a meaningful picture is illustrated of just what goes where in the form of a dataset. It is simple but I think it is detailed here just what it can mean to draw from code in a meaningful and mathematically wise fashion. NDepend bases their algorithms on such ideas as this. It may not be exactly what I want out of their software but what is great is they actively avoid the mistakes associated with LOC. Using it to compare developers or to motivate developers is dismissive. There are an abundance of ways in which this can be done otherwise, and many open source alternatives set out to accomplish.

- Observing branching and branch strategies.

- Comparing builds and deployments through all environments.
- Static code analysis.
- Patching or updating servers and more.
- Responding to information security requests and more.

And this would be pertaining more to the DevOps side ultimately, but I think a lot can be said about how many processes exist for this discipline outside of simply programming. Maintaining and updating various things is an unusual part of a project in more ways than I could name off the top of my head or even with references. The truth is unless such things are accounted for even LOC will start to be a disingenuous measure of productivity even in the very long term. There needs to be discussions about just how platforms can simply get better information and help programmers, not project managers.

2.3 Setup for analysis

With datasets readily available by API and repository information [6] [5] it is easier than ever to add to a database with what we need when we need it. It is important to hold LOC and its impact close to heart so I think line additions and removals are important data points. But for extra measure I would also apply my own inspiration and associate names to every individual commit and the code changed to better understand who is deleting what. I think on a macro level this could have better implications used in an algorithm than LOC.

Aside from this I think project tasks and more should encapsulate these commits to categorize data collected. It can supplement the measurement of sprint burndown and other sprint to sprint metrics. Meanwhile branching from repository data will help to locate developers as well as manage the coinciding of developers as the cooperate. Weighting tasks with a specialized unit will be easy to base off of various bounty related metrics so this data is absolutely important to the creation of a smart evaluation system.

Chapter 3

Computations

3.1 Evaluating bounties

I personally think a system involving bounties benefits developers already in many differing forms. To potentially compute what I think would be a wholly unique coefficient for prioritizing however draws from elsewhere. The way it should be seen is that a developer will over time hop around to new places to work on code based on their performance over several initial data points or in other words features and tasks. The comparison made between those initial data points will help select starting traits for this developer and constitute where they ought to try and make implementations next sprint or cycle.

This leads me to think that an algorithm for determining relevance based on tags like with search engines would optimize everything akin to the search engine boom two decades ago. Research [10] shows how query models and document models deal with the relevance problem and how it essentially replaced any other prevailing ideas prior. Relevance is not tied intricately to ranking, and ranking is just how we eventually display the results. By computing a relevance score for people and tags for repositories to slap onto their respective profiles it can be made extremely clear what they are showing engagement or increased performance on. Accounting for the likes of project details would of course factor into a function for calculating new relevancy scores.

3.2 Counting lines

With reference to the lines of code discussion [8] it should be said that the methods employed to average out production estimations are revolutionary and nothing short of that. But by counting lines removed a correlation can be made between users that have social implications. Are users working with each other? What do their interactions say about their philosophies with regards to programming? Was deleting code warranted? Perhaps among these questions is why is code being removed that was already made in the first place.

Although not all these questions could possibly be answered satisfactorily I think this problem could be reduced to a simple computation of "lines identified as redundant by user". The identification of redundant code is not always to me a productive use of time and for various reasons.

- The code was someone's own and it was changed not long after being written. This removes the legitimacy of the lines pretty quickly and should be acknowledged as such perhaps by marking the initial code as ignored. If it was instead replaced much later after a sprint then there is reason for it to be a reflective change and the result of engaging with the engineering process. This allows for less replacing overall and more self reflection. In hopes that it would cut some slack for developers under stress and conscious of their code, the computation of warranted code revisions would be a great way to be understanding of developers and their habits. In some respects it would be therapeutic and increase accuracy of LOC computations discussed by NDepend associates [8].
- The code was made by someone else but replaced soon after. This would say a lot because there is a clash between two users evident. If a user changes code they essentially take away work from another user. Should it be counted as theirs now? The answer would be yes and no. If marked on a curve, too many code replacements could be penalized for unnecessary stealing and intrusiveness with regards to the measurement process. Therefore any poor code replaced can be discovered as it was encouraged that developers do not interfere with their partners. Itemizing individual work would be key here and greatly motivate developers in the long run. In conclusion anyone replacing too much is acting against their own interests while anyone getting replaced too often will eventually be found out for why they were being replaced if for good reason.
- After a long while code from someone else was replaced. This indicates something of a mixture of the last two points: that if people do not revise their code after some time and someone else gets around to it some light penalty may be warranted. It would not punish users so much as it would get a better idea of what code written was relevant to a developer at the time. It could help course direct other parts of the system to finally get new data on old information and become more accurate in retrospect. On a larger scale many of these assessments could amount to an entire new vision for the distribution of development tasks.

3.3 Positive engagement

Calculating engagement with data points or other developers via "social links" will naturally draw from areas such as the relevance problem [10] and other discussions already referenced in this chapter. The distinguishing factor though will definitely be that here I elaborate on the final realization of software engineering capability. The representation would be in taken in the form of aggregating achievements according to how they are ranked based on developer qualities found in their relevancy tags. Basically it would be all about giving precedence to developers with positive engagement with more important projects. To clarify positive engagement does not imply any developer is better than the others, although some aptitude will naturally seep into such computations.

Rather the cornerstone of the initiative is to identify where people need attention. The computation involves constantly prioritizing who to juggle upwards and how to give them new work. This will involve identifying whether positive engagement is as a result of the match ups alone. It can then through algorithms be tested as a hypothesis to see if new avenues can be discovered for suffering software engineers. At its core its a booking system. This would be at the centre of most of these google coding cooperation contests for students with open interpretation matching problems. Google Hash Code [11] tasked students to optimize scores for certain conditions most years. Their outline of a problem looking at specific times and essentially booking algorithms to get the best outcome is what I think could in an essence derive an equation for positive engagement in the project.

3.4 Rating friendliness

On a very similar note I want to discuss the interactions between developers. How developers treat each other's code and coincide with each other in general can help to make interesting deductions. I mentioned earlier about social links as a concept for developing the relationship between two participants in a model. It should be considered the weight of these links and how it affects preconceptions about code changes. It should be acknowledged in light of what has been discussed so far that replacing someone's code has a whole new meaning if two developer's have trust in one another. It should actually be rewarded for good practice and a valid progress of software engineering development.

This concept will entail the revision of the 'counting lines' section earlier but more as an addition not as a complete evaluation of it. It is true that at the start developers will have their own motives separate from what they want to contribute, but it could be said that once developers contribute a certain amount on the same feature together they have gained simultaneously adequate understanding of the feature. As well as this they both likely have experience - if they both have experience working together on the feature their actions are of more importance in this space.

The most beneficial way to account for this without doing a complete turnaround from the curve based model, replacements should be scrutinized for testing improvements or reductions. In essence with itemization of features it should be noticeable when it needs to function better and if it actually meets this standard. Otherwise replaced code does not reduce the replaced code's value but still raises the value of the new overlapping code.

3.5 Finding laziness

While no system is perfect and even my own computations are limited in scope, to reinforce them with feedback is essential to give the illusion of an ever expanding openness to innovative techniques. If a person were to after everything gather bad data behind them and become suspect of not performing well, it should be expected there are exact reasons they resulted in poorer products in and showed visible patterns. The blessing of having undeniable proof is that a user must not necessarily be constantly upright but just modest and attentive. A system I would have in mind favours understanding and strives to improve for the sake of the users upon which the entire project relies.

To find true laziness in software engineering measurement it would not involve much computation beyond trivializing code which is a piece of cake for AI even at present. Research into machine learning makes it possible to do some tasks to a certain degree with exceptional accuracy. If by this method a team can reveal how predictable a member's progress has been then there is no doubt their work is not original or their own. Such a method would be a last resort and rely on several factors that come into play after a suspect is found potentially guilty of not contributing in any significant way to the product.

But I find this is a great backbone for the heavier maths used to find success. To simply find the dichotomy between complacent and competent software engineers enables managers to find the weeds so to speak. The rise of low code artificial intelligence could almost resemble a poor software engineer with no engineering and just programming to distinguish their work [12]. While the technology gets low adoption from developers despite the hype surrounding it according to Gartner, this is irrelevant to the purpose it could fulfill in an already budding industry in the education technology sector in popular plagiarism checking software. Essentially the idea is through even the likes of a simple markov chain to evaluate the likelihood of pointless code. The opportunities for such evaluation at the jobs level could be a great filter of low performers.

Chapter 4

Ethics

4.1 Scrutiny

The fall backs exist when it comes to such elaborate schemes as those I have proposed. But I think with being watchful rather than overly observant of developer actions there is a better focus on finding out if a modest amount of work is done while not just assessing LOC as the metric to end them all like any manager using simply analytics on GitHub repositories.

Some assumptions will have been made in metrics that look into how code is replaced or how organization skills are necessary to even maintain a reliable task board for project management and then automatic data analytics on that data, so it goes without saying without good trials of some product it is only about as efficient as some of the services discussed currently being used today. In theory I believe I left no room for abuse in the system if run right.

4.2 Intrusiveness

Like with any system that intends to gauge employee interest and provide automatically generated feedback monotonously, there are some camps that would be immediately against reading into every virtual movement on web apps as well as company repositories. To some extent this would involve intrusion into personal work life balance. All complaints with these concerns are completely valid and actually provide great feedback for these systems to function better. I consider mine very pervasive but only where it matters and more spread out as to focus more on general values. In that way I think there are generally less concerns with my system that prioritizes finding the right job for everyone instead of ranking performance for no reason other than to punish perceived negligence which benefits nobody. Only as a bonus does it really gather proper statistics about employee efficiency working in a team, which is a great side effect and in my opinion a lot more indicative of performance than a LOC coefficient in a graph and only surface level incorporation of the agile method.

Bibliography

[1] Mark Abadi. “Jeff Bezos once said that in job interviews he told candidates of 3 ways to work — and that you have to do all 3 at Amazon”. In: *Business Insider* (Aug. 12, 2018). URL: `https://www.businessinsider.com/jeff-bezos-amazon-employees-work-styles-2018-8?op=1&r=US&IR=T?utm_source=copy-link&utm_medium=referral&utm_content=topbar`.

[2] Renee Johnson. “Harness the Power of Project Management Tools”. In: *The Tech Report* (Dec. 8, 2021). URL: `https://techreport.com/software/3475208/harness-project-management-tools/`.

[3] In: *Agile Manifesto* (). URL: `https://agilemanifesto.org/`.

[4] Martin Fowler. “The New Methodology”. In: *MartinFowler.com* (Dec. 13, 2005). URL: `https://martinfowler.com/articles/newMethodology.html`.

[5] In: *Trello Docs* (). URL: `https://help.trello.com/article/756-trello-api-documentation`.

[6] In: *GitHub Docs* (). URL: `https://docs.github.com/en/rest/reference/teams`.

[7] In: *Gartner* (). URL: `https://www.gartner.com/en/newsroom/press-releases/2017-10-02-gartner-survey-of-more-than-3000-cios-confirms-the-changing-role-of-the-chief-information-officer`.

[8] In: *NDepend Blog* (). URL: `https://blog.ndepend.com/mythical-man-month-10-lines-per-developer-day/`.

[9] Frederick P. Brooks Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison Wesley, 1975. ISBN: 0-201-83595-9.

[10] In: *moz.com* (). URL: `https://moz.com/blog/determining-relevance-how-similarity-is-scored`.

[11] In: *towardsdatascience.com* (). URL: `https://towardsdatascience.com/google-hash-code-2020-a-greedy-approach-2dd4587b6033`.

[12] In: *techmonitor.ai* (). URL: `https://techmonitor.ai/technology/ai-and-automation/low-code-ai-robotic-process-automation-api`.