Measuring Engineering - a report

To deliver a report that considers the ways in which the software engineering process can be measured and assesed in terms of measurable data, an overview of the computationial platforms available to perform this work, the algorithmic approachs availabe, and the ethics concerns surrounding this kind of analytics. Reading list available here. 10 pages.

1. **What data**
2. **Where to compute**
3. **What algorithm**
4. **Ethics**

What data can one gather that truly measures a software engineer’s output? How can a highly educated indivdual be analysed when the majority of their work is done in the inner thought processes of their head? A software engineers aim is not to produce the most lines of code, or get the job done the fastest but to provide the best solution possible to the problem at hand. How can you measure this?

An answer to the question “How should a software engineer's productivity be measured?” that reallly struck me was Ori Shalev’s on the website Quora. He replied:

*“It can't be measured, because what you refer to as “productivity” presumes a commonly known path of progress for a task, whereas true greatness is in finding the best path - not following an existing one in faster pace.*

*You can measure drivers on how efficiently they use fuel when driving from point A to point B. A truly great driver could find shortcuts - such as boarding a ferry to cross a bay. That sounds very uncommon, but in software engineering it happens all the time. You would never know if an engineer that you don’t have on your team would solve the same problem as yours in 20% of the effort and 2x the robustness. Measuring productivity would be a distraction from focusing on getting the most value out of your team.[[1]](#footnote-1)”*

Many who attempt to analyse a software engineer’s productivity forget that these people are at the forefront of innovation. They are artists. If one was to measure an artist such as Leonardo Da Vinci’s productivity, how would you go about it? The number of brush strokes a day? Rebecca Elfast is a Swedish painter who is renowned for the her use of as few brush strokes as possible.[[2]](#footnote-2) Does that mean that if I layered multiple coats of paint over a canvas my art is more productive, or better than hers? Certainly not! The same applies to software engineering, the old standard of using the number of lines of code per day cannot be used any more as one line of a high level language could take 3 days to write and debug, simillar to how one brush stroke could take a whole day to paint.

Another reason why past attempts of measuring output through ways such as lines of code, bugs fixed, stories closed, passing tests failed is because engineers are smart. Clever engineers can easily manipulate the results to make it appear that they are doing more than they actually are. A salesman cannot easily lie about how many sales he made that day as the cash register will present the true value, where as it is easy to space code over a number of line, rather than condensing it down into one or two. Why use a loop when you are being measured on the number of lines of code you produce? There’s a quote about this commonly attributed to Bill Gates:

“Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs.”[[3]](#footnote-3)

If you follow SCRUM as your software development methodology, you should be able to estimate with a decent level of accuracy after a few sprint cycles, what's the individual velocity of each software engineer, and use that velocity as a productivity metric.

It’s infamously impossible to measure the productivity of a software engineer (or a software engineering team), because there’s no way to measure the output of a software engineer.[[4]](#footnote-4)

Without a measure of output, we can’t know whether you’re a more or less productive engineer than you were last year, or whether your team is more productive than it was last year. We can’t decide whether I’m more productive than you are, or whether your team is more productive than my team. We can’t decide whether to change programming languages, whether to adopt “agile” methodology or reject it, or whether to change the way we interview candidates.

But, intuitively, such variations in productivity must exist. For example, as engineers, we often need to change our code and then re-run it to see the results. If we made your compile-and-refresh loop take ten times as long — or a hundred times as long — surely that would impact your productivity.

And yet, we can’t measure how much productivity would be impacted. If it takes us ten times as long to compile/refresh, maybe that’s a critical “hair on fire” bug, or maybe it’s just mildly annoying but harmless to productivity, or maybe it would paradoxically even improve productivity, forcing you to take a break to refresh your mind and spirits.

It turns out that that if you ask engineers what they like/dislike about their jobs, the things they’ll complain about seem to have something to do with their productivity. They’ll tell you about long compile times, having too many meetings, or a noisy work environment. They’ll ask for another big monitor, clearer specifications, or more time to pay down technical debt.

Across all industries, satisfaction at work comes down to feelings of autonomy (directing our work), mastery (accomplishment and self-improvement), and purpose (working on something more important than yourself).¹

Unproductivity is the opposite of mastery. It attacks job satisfaction directly. But, unlike productivity, we can measure job satisfaction.

Use Net Promoter Score to measure employee satisfaction

Redfin is both a real-estate brokerage and a software company. (It’s weird, but it’s a good weird. We’ve even adopted a slogan: “Keep Redfin Weird.” It works for us.)

We run our real-estate business using a standard metric for customer satisfaction, the Net Promoter Score (NPS). NPS was designed to measure customer satisfaction, but you can use it to measure employee satisfaction as well.

If you haven’t heard of NPS, it works like this. We survey each of our customers with this question: on a scale from 0 to 10, 10 being highest, how likely are you to recommend Redfin to a friend? A 9 or a 10 rating on that scale is called a “promoter”; 7 or 8 is called “passive,” and 6 or below is called a “detractor.” When computing NPS, you take the percentage of promoters and subtract the percentage of detractors. Thus, NPS ranges from +100 (everyone’s a promoter) to –100 (everyone’s a detractor).

We use NPS to manage the performance of individual real-estate agents and their managers; NPS is the single largest factor in our agents’ compensation. Traditional real-estate agents are paid solely on commission—a percentage of the sale price—which means that the agent and the buyer’s interests are conflicted. The more you pay, the more your agent gets. Redfin’s agents are salaried employees who receive a bonus for high NPS scores. The data shows that our approach delivers better results than traditional real-estate agents.

We’ve had fantastic results measuring customer satisfaction with NPS, so we naturally decided to measure the satisfaction of our employees the same way, with the same basic question: “How likely are you to recommend working at Redfin?”

We measure NPS in every team in the company, but our engineering management, starting from our CTO, Bridget Frey, focuses on the NPS of our engineering team. We do that partly because we want to recruit and retain the best employees (which is a huge challenge in today’s job market), but also because we believe that the NPS of our engineers is the closest metric we’ll ever have to a measurement of their productivity.

NPS is the best measure of engineering productivity

If you’re an engineer reading this, you’re probably skeptical, and I say that because so many of the best engineers I’ve worked with maintain a healthy sense of skepticism at all times.

I readily admit that we can’t prove that NPS is the best measure of engineering productivity, or even a good measure of engineering productivity, because that would require us to directly measure engineering productivity, which is impossible.

But let me toss a few brief arguments at you in favor of measuring and improving employee NPS, in the hope that one of them sticks to your face.

I predict that if you survey your engineering organization about what makes them unhappy at work, the vast majority of the issues they’ll report are factors of unproductivity. Don’t take my word for it; just try it.

I also predict that you’ll find, as we find, that if you focus on your engineers’ top complaints—things that they say are driving down their productivity—then you’ll find that the satisfaction of your engineers rises.

Measuring employee NPS doesn’t mean we have to stop recording other metrics, such as bug counts, user retention, or conversions. That stuff still matters. But when asking, “How do we measure whether our engineers are more productive?” we’re looking for the least confounded metric. A zillion factors can confound the NPS measurement of productivity, but other metrics are even more confounded. By “best measure” I mean it’s the “least bad measure.”

If we grant for the sake of argument that unproductivity doesn’t cause dissatisfaction, I argue that satisfaction causes productivity, and that dissatisfaction causes unproductivity. I may not be able to measure my productivity, but I know that I feel a lot more productive when I come into the office in a good mood, excited about the job. This creates a virtuous cycle, where improving productivity improves satisfaction, which improves productivity, which improves satisfaction.

In the worst case, you might “waste” a bunch of time/money improving the satisfaction of your engineers for no productivity gain at all. (Aw, darn!) But employee NPS is a good leading indicator for employee attrition, which is a huge and quite measurable cost to any engineering organization. It’s also a good indicator for how many candidates your employees will actually refer to you, reducing the costs of recruiting.

Finally, if you’re an engineering manager, I hope you believe, as we do at Redfin, that the purpose of management is to nurture employees, not to rule them. In that case, NPS is a direct measure of the productivity of engineering managers. Judge your own mastery of your work by the satisfaction of your team. One way to improve team morale, perhaps the easiest way, is to fix the factors of unproductivity that they identify to you. Listen to their feelings of unproductivity and address them. It’s worth it.

Now the question boils down to how we should measure ourselves.

The most obvious measure is productivity. But what does that mean? Lines of code? Every decent developer knows that reducing line count is often better than increasing it. Even in the early eighties there were misgivings to this approach.

How about hitting GA dates? There are two big problems here. Setting a date months out implies a waterfall approach which for a dynamic startup is generally not the best approach. Secondly, and more importantly, anyone can hit a date at the expense of quality.

Hmm…so what are we to do?

The conventional wisdom is that Engineering teams can not be measured effectively so we shouldn’t even try.

Martin Fowler thinks it’s a fools errand: “I can see why measuring productivity is so seductive. If we could do it we could assess software much more easily and objectively than we can now. But false measures only make things worse. This is somewhere I think we have to admit to our ignorance.”

Joel Spolsky agrees: “Let’s start with plain old productivity. It’s rather hard to measure programmer productivity; almost any metric you can come up with (lines of debugged code, function points, number of command-line arguments) is trivial to game“

While I hold these two legends of our industry in high esteem, I respectfully disagree. Software engineering teams CAN be measured… it’s just a little complicated.

The Building Blocks of a Great Engineering Team

Let’s examine the components of a high functioning team:

Business Impact: This is what it’s all about! Great running code is useless without business impact. But get a little more specific. For most real projects you’re building new software features for a specific purpose: increasing the number of users, driving better adoption of the system, etc. These should be the most important goals of the product management team, but a healthy engineering team is highly aligned with the product team, and thus should be thinking about these goals.

Productivity: This is probably the most controversial. Many people in the agile community will argue that productivity shouldn’t be measured as it’s counter to the agile philosophy. You should instead track velocity to get an idea of when things will be delivered. I disagree here. We have two week sprints iterations1 at Kapost, and based on historic velocity we have a decent idea of what we can get done in an iteration. But as I often discuss with my teams it takes super-human focus to get things across the finish line. And that is usually not conveyed in user stories. So each iteration we set concrete high level iteration goals (based on velocity). A simple example is “this feature will be shipped to production.” This removes any potential gamification of story points, and keeps us focused on what actually matters, delivering customer value.

Bugs: There are two dimensions to this: volume and severity.

Volume: How many bugs do we have at any time? We have three different priority levels for bugs, and we set thresholds for each level. We measure ourselves by how well we’re keeping to the thresholds.

Severity: How many users are reporting bugs? A single bad bug will be reported by many users. Severity should be measured relative to the number of active users. eg 10 bug reports from 1mm active users is outstanding, whereas from 10 active users is terrible. By looking at both volume and severity you get a good understanding of the state of bugs.

Code Quality: Again a couple things we measure here.

Test Coverage: We use various tools for Ruby & JavaScript to determine if our test coverage is where we want it.

Code Quality: We use CodeClimate to give us a grade on each of our repos. This also gives us insight into pull requests and what changes should be made before we merge code in.

App Performance: Last but not least is the performance of the application. The only things we care about here are user facing applications/apis. We measure using Apdex via NewRelic, which does a good job of measuring user satisfaction. To keep it simple you could also just measure average page load time or even better 90th percentile. This is however trickier than it seems as a modern web app doesn’t really have a concept of a “page load”. We’ve instrumented our frontend router to measure key events instead.

Putting it All Together

I’d argue that if your team has high achievement of each of those components then you’ve got a high functioning team. At Kapost we set goals for each of those components each quarter. For some teams and some components were already doing well (hurray!), and we put in a “threshold goal” which means we’re just trying to maintain that level. In other areas we want to improve so we set a goal of improving a number by the end of the quarter.

Then we measure ourselves every iteration. Improvement goals are split up over the course of the quarter, so for example if we’re trying to get our code climate score for a particular repo up from 3.3 to 3.6, the goal for the first iteration is 3.35, then 3.4, etc.

Finally we weigh each of these metrics to give us a blended score. These weights vary from quarter to quarter depending on what we want to focus on. This is the current blend.

The average of the score for each iteration is your final score for the quarter and this is what we base bonus payouts on!

We’ve been doing this for about a year, and it is working great so far. While the blended metrics take a little bit of effort to measure, with a clear dashboard every team knows how they’re doing, what needs improvement, and the whole company knows how we’re doing as an engineering organization.

1 I personally detest the term “sprint”. Sprinting through an iteration is a great way to burn out a team. A healthy team is capable of high output for an indefinite amount of time. It’s much more akin to a marathon than a sprint. Thus we use the term “iteration”.[[5]](#footnote-5)

If there is a holy grail (or white whale) of the technology industry, especially from a management standpoint, it's the measurement of developer productivity. In fact, there is a very common phrase, "you can't plan if you can't measure." Measurement works so well in many other industries that involve humans -- building construction, manufacturing, road work. We are able to get rather accurate estimates for both cost and completion date, so why not software?[[6]](#footnote-6)

Why Measure?

We as developers love to play along with this. So much of what we work with is data-driven feedback. We can analyze with profiling, complexity, conversion rates, funnel metrics, heat maps, eye-tracking, a/b testing, fractional factorial multivariate analysis, etc. All of these things give us data upon which we can prioritize future efforts. It only makes sense that we should be able to measure ourselves.

Measuring Developers

Measuring and managing developer productivity, however, has consistently eluded us. So many of the tools we use are designed to increase developer productivity: XP, TDD, Agile, Scrum, etc. There were academic papers analyzing software project failures/overruns in the 80s. This isn't a new phenomenon by any means. We also famously hear of IT failures in the news, such as:

(2004) - UK Destroys Tax Records, costing at least £85m.

(2004) - Ford and Oracle scrap Purchasing System, costing $400m

(2007) - FBI Virtual Case Files Scrapped, costing $170m

(1962) - Rocket Failure for Missing Hyphen, costing $135m in today's dollars.

These are just a few cases. There are likely dozens or hundreds of errors on this scale every year, and likely hundreds to thousands of projects in the <= $1m range. A lot of this is due to a lack of good testing. We at Dev9 have frequently espoused the benefits of automated testing, and it has real benefits.

However, quite a few others are caused by planning and estimation that missed the mark. There are estimates that say IT organizations will spend over $1t per year on their IT initiatives. Notice it's trillion, not billion. A trillion dollars. Given this extremely high cost, anybody who found a way to reliably gain efficiencies of even 1% would save a billion ($1,000,000,000) dollars. That's a lot of zeroes.

The 10x Developer

There is a theory floating around, and largely backed up by data, that the best developers among us are 10x more efficient than the worst ones. Given that developer salaries do not reflect this order-of-magnitude difference (Who is the last senior dev you knew who made $800k/yr?), it's obviously a bargain for companies if they can find one of the 10x, and hire them at a comparable rate to a 1x or 2x person. These studies even gave birth to analysis that showed, "...[T]he top 20 percent of the people produced about 50 percent of the output (Augustine 1979)." If you were a manager looking to cut costs, you'd want to get rid of 80% who produced only 50% of the output, and hire only the kind of people who are in that top 20%.

High Performers

However, that quote I gave you is not the full quote. It actually is, "This degree of variation isn't unique to software. A study by Norm Augustine found that in a variety of professions--writing, football, invention, police work, and other occupations--the top 20 percent of the people produced about 50 percent of the output, whether the output is touchdowns, patents, solved cases, or software (Augustine 1979)."

This problem is not a software-specific problem. Any field that requires human decision-making is subject to variation. Some people are going to be naturally talented in the field. Some have the perfect personality for the job. Some people are voracious readers, others never try to learn after school. Some consistently push their bounds, while others are content to be competent. Some people's brains just work differently. Some people's bodies just work differently. It doesn't take a genius to see that some football/soccer/hockey players are dramatically better than others, even though they both train the same amount of time. Why would software development be any different? Why should it?

Traditional Measures

Before we continue onward, let's look at some of the ways the industry has tried to quantify development activities, and why they fall short for measuring productivity. The tl;dr of this section is that any metric you come up with to measure developers will be gamed.

Hours Worked

This is one of the most obvious ones: butt-in-seat time. If you worked 10 hours instead of 8 hours, you should get 125% of the work done. That's just math. Time and time again, you'll see studies proving that this just does not work for anyone. In fact, running hot on hours is a great way to decrease productivity.

The Relationship Between Hours Worked and Productivity (Stanford)

Henry Ford Drops Hours, Increases Productivity

Stop Working More Than 40 Hours Per Week

Time and time again, we see proof that more than 40 hours necessarily leads to a drop of productivity, even for assembly line workers. Yet, this pervasive attitude of 8-6 being a minimum workday continues to chug along.

I was once on a team where the managers were so addicted to tracking hours as a measure of productivity that we started putting meetings, lunches, and bathroom breaks on the board every sprint. Otherwise, we were accused of not working hard enough because our hours didn't exactly add up to 40 or more. This absolutely destroyed the morale of the team. "Don't forget to put your hours in" causes me to involuntarily twitch.

Source Lines of Code (SLOC)

Lines of code. What a perfect measure. Even if they think different and whatnot, we can just track lines of code, and use that to extrapolate.

There are so many problems with this metric that it is actively harmful to use it to judge developers:

Developers can just add extra lines of code to pad their numbers

A 200-line solution may be faster or more performant than a 1000-line solution to a problem

Sometimes the solution is to delete code

5000 lines of buggy code is worse than 1000 lines of bug-free code.

Developers copy-paste code instead of refactoring, leading to massive technical debt and poor design, as well as significantly increased bug probability.

This is an interesting metric to track in aggregate to get a sense of the size and complexity of the system, but not useful at an individual level.

Bugs Closed

This one is so crazy, Dilbert has a comic on it:

Dilbert

If you do this, you're the pointy-haired boss from Dilbert.

Function Points

Function points found a small following out in the world. You've probably never heard of them. It's practically impossible for a lay-person to digest. If you want to try to measure function points for your project, then give this article a read and figure out how to automate it in your project.

Go ahead, try it. I dare you.

Defect Rate

The idea of this one is to measure the number of defects each developer produces. This does seem reasonable, and you should probably track it, but here's why it's a bad measure of productivity:

It favors bug fixes over feature development.

It discourages developers from tackling larger projects. Would you rather try the "Add a form field to this existing page" project, or the "Implement a real-time log analysis system from scratch" project?

Not all bugs are created equal:

Bug 1: When somebody uses the "back" button, a bug deletes all customer data on the production website.

Bug 2: Form fields are not left-aligned

Bug 3: If a customer enters dates that span 2 leap years, the duration calculation is off by 1 second.

People often mistake features for bugs. Missing requirements are not a bug, but may be filed as such.

There may be multiple bug reports related to 1 bug.

Developers will never touch anybody else's code, and will get very aggressive about protecting their code.

Defect rates are interesting, but they're not enough to give you an idea of productivity.

Accuracy of Estimation

Estimation, my least favorite activity. I have no problem taking a swing a how long something will take. However, at every single company I've ever worked for, estimates become commitments. If you say "this will take about 3 days," you get in trouble if it takes longer than 3 days. On the other hand, if you finish ahead of schedule, you get praised. This encourages developers to estimate given an absolute worst-case scenario. Like, "neutrino streams from solar flares corrupting random bits on our satellite stream that somehow passed checksum validation but is still corrupted and we wrote that to our hard drive" kind of worst-case scenarios.

Other reasons this metric is a problem:

If you estimate in "ideal hours," distractions may turn that 8-hour task into 3 days.

Developers can be overly and inconsistently optimistic with their estimations.

The scope was not adequately defined, or not defined at all.

The customer was asking for something that is impossible, which could only have been discovered at coding time.

There is one more reason, bigger than those four combined. Look for the section "Developer Productivity is a Myth."

Story Points

Story points -- we thought we had found the holy grail. Story points were explained as a measure of effort and risk. If we have consistent story points, and figure out how many story points each developer finishes per sprint, then we can extrapolate developer performance. Let's see what happens:

If they finished less than they did last sprint, they're chastised. They are again reminded that they committed, no matter what. Even if you had to help a prod issue, or were in a car accident, or got sick -- you committed. So developers start sandbagging to avoid this.

If they finished exactly right, the managers will think the developers finished early and were sitting idle, or were padding their estimates. This leads to frustration and resentment. Alternatively, a perfect finish might be seen as a state where, if everybody worked a few more hours, we'd see more output.

If they finish with more points than they took on, managers will accuse the developers of sandbagging. Then they told that they must accept more points next sprint, to take this into account. That, or you have a "level-setting meeting" where everybody re-agrees what the points represent. This leads to frustration and resentment, not to mention the drop in productivity related to figuring out the new point system.

If a manager asks for doubled productivity, that's easy: double the story-point estimate.

Story points also aren't consistent between developers. Even if everybody agrees that it's a 3-point story, based purely on effort and risk, the wall-time delivery will be different depending on who picks it up. One developer who is intimately familiar with that code may be able to finish in 2-3 hours, while a new junior developer may struggle for 1-2 days. This is proof that we've decoupled productivity from points, and why it's a bad metric.

On the official Scrum forums, practioners always have to explain why story points are not a measure of productivity. The Scrum Alliance even has a whitepaper called The Deadly Disease of Focus Factors, and here is the opening statement of the document:

To check your organizational health, answer these two questions:

1) Do you estimate work in “ideal” hours?

2) Do you follow up on your estimates, comparing it to how many “real” hours work it actually took to get something done?

If so, you may be in big trouble. You are exhibiting symptoms of the lethal disease of the “focus factor”. This is how the illness progresses:

Speed of development will keep dropping together with quality. Predictability will suffer. Unexpected last moment problems and delays in projects are common. Morale will deteriorate. People will do as they are told, but little more. The best people will quit. If anything gets released it is meager, boring and not meeting customer expectations. As changes in the business environment accelerate, the organization will be having trouble keeping up. Competitors will take away the market and eventually the end is unavoidable.

So even the people who invented the concept tell you explicitly not to use story points as a measure of developer productivity. So stop it.

Developer Productivity is a Myth

"You can't plan if you can't measure." This is an idea still taught in business school, it's a mantra of many managers, and it's wrong in this context. It assumes everything a developer does is objectively and consistently measurable. As we've shown above, there still doesn't exist a reliable, objective metric of developer productivity. I posit that this problem is unsolved, and will likely remain unsolved.

Just in case you think I'm spouting nonsense, just remember: the smartest minds of Microsoft, Amazon, IBM, Intel, Wall Street, the Bay Area, Seattle, New York, and London still haven't found that magical metric. It is, therefore, a rather safe assumption that the average company also hasn't found it. If you believe you have proven me (or them) wrong, go ahead and publish it. You'll be a wealthy rockstar of the programming universe. People will write books about your life and your brilliance.

We all know that some people are better than others. Developers can identify which developers are better, but there is not a number or ranking system we can come up with, objectively based on output, that consistently and reliably ranks developers. Let's explore why.

A Developer's Job

Most people don't understand what developers do. We clicky-clack on electronic typewriters while drinking Mountain Dew and eating Doritos in the dark, and make the magic blinky boxes show cute cat pictures.

OK, it's not the 90s anymore. Most people really do understand the basics of operating a computer. If you're under 40, there's a good chance your grandparents use Facebook.

So what do we do? Code is the output, but it's not really what we do. If we were just transcribing something into code, that's basically data entry. We're knowledge workers. We take inexact problems and create exact solutions. Imagine if managers were capable of exactly specifying the system they want built. They would have to explain it so finely-grained that it would be programming. That's what we do. We are people who exactly detail how a system works. Our code is the be-all, end-all specification for what the software does. We are people that write specifications, digest knowledge, and solve problems.

Most people are incapable of breaking a problem down to the level required for computer code to solve it. This isn't to say that they can't learn, but it's a skill you must nurture. Imagine a parent (P) trying to teach a kid (K) how to make a grilled cheese sandwich:

K: How do you make a grilled cheese sandwich?

P: You make a cheese sandwich, then fry it in a pan until it's done.

K: What's cheese?

P: It's a food made from milk.

K: How do they make cheese?

P: Well, they take milk, and they add rennet, then they add flavorings, and maybe age it.

K: What's rennet?

P: It's an enzyme that makes the milk solid

K: How does it do that?

P: It is a protease enzyme that curdles the casein in milk.

K: How does a nucleophilic residue perform a nucleophilc attack to covalently link the protease to the substrate protein before releasing the first half of the product?

P: Because I said so.

Imagine the plethora of questions they can keep asking: How do you tell if it's done? What does done mean? How many minutes? What's a minute? Why is a second a second and not something else? How brown is too brown? What kind of bread do you use? How do you make bread? What is bread yeast? What's butter? What's a pan? How do you make a pan? What's a stove? Why does a stove get hot? How does a stove get hot? What happens if you don't have cheese? What happens if you don't have bread? Can you use a microwave? Can you cook it outside if it's really hot? Can you use other cheeses?

So when somebody in the business asks, "can you tell me how many people visited our site yesterday and clicked on the newsletter signup?", it sounds like a simple request. You just take all the people, find the ones who clicked the thing, and count it. But, let's take a dev perspective. How do we identify visitors? Is IP good enough? Do we support IPv6? Do we want to use cookies? Is our cookie policy legally compliant in Europe? Do we have to worry about COPPA? Do we want to de-dupe visitors? How do we track that people clicked on a link? What's the implication of click-stream tracking? Will our infrastructure support that? How important is accuracy? If we lose one click record, does that matter?

This is what developers do. For every line of code we write, we are answering all of these questions in excruciating detail.

When you hear developers talk about "abstraction," we are basically answering the "How does electricity get turned into heat?" question for anybody who asks. Then we're answering the "how does a protease enzyme curdle casein?" question. Then we're answering the "how does heat turn bread brown?" question. One of the questions we literally answer is, "How do you turn 1s and 0s into text?" Well, what about character encodings or code pages or multi-byte entities or byte-order markers or little-endianness... you get what I'm saying. A computer is a dumb machine. It can't read our minds, and has no context.

A good developer is able to take a high-level problem, see best way to break it down, and create the correct levels of abstraction, all while keeping the code readable and maintainable for other developers. This also explains why some people are 10x performers, and some people get so frustrated with programming that they give up. Some people have curated, or have a natural talent for, thinking at this extreme level of detail. Some people can intuit things that others will never discover -- even if they had all the time in the world. This is the nature of knowledge work.

Professionals

This one is likely to be more controversial, but the crux of this issue is that developers are treated like blue-collar workers. Because so many of our beloved processes come from the world of manufacturing, it's very easy to see why developers would be though of like assembly line folks. That's why managers try to get consistent productivity. The idea is that if they can just find a way to measure developers, then developers will truly be interchangeable cogs: software would never be late again, it would always be on budget, and it would be exactly what we want. All of the theory they learned about manufacturing and assembly lines in business school would then apply to this field.

This attitude led to the massive amounts of off-shore outsourcing, just like manufacturing. These days, we know that offshore development is very difficult to get right, the end product often contains a lot of bugs, and is often of very poor quality. Many companies are bringing off-shore projects back in house due to these issues -- or using local consulting firms like Dev9.

What About Those Builders?

So what makes building and road work so predictable, when we can't get it right for development? The answer is relatively simple: we're not doing the same job. The labor in those fields have very little input on the decision-making processes. As we explained above, what a developer does all day is make thousands of tiny decisions. By the time these construction projects break ground, the decisions are made, the plans are already in place, there are very exact specifications, and there is little room for ambiguity or disagreement. In addition, the skills required aren't as widely variable. One person can use a pneumatic nailer about as good as any other. One person can operate a dump truck about as good as any other. And even if somebody was a 10x better paver than another, the time needed to cure is a near-constant factor. In addition, the tools and techniques are not as rapidly moving. The basics of foundations, jack studs, jamb studs, nogging, top plates, mudding, and taping really hasn't changed. Governments and building codes will dictate many of the decisions, like how far apart studs are center-to-center, or how many electrical outlets go on a wall.

Rather than trying to build an analogy to builders, who makes all the decisions? City planners, building code authors, architects, and engineers. All while dealing with a highly beaurocratic permit system, and localities that have different rules. They make tons of decisions.

Professionalism

Let's do another thought process. If developers were truly thought of as professionals, let's see how other professions compare.

DOCTORS

Ask a doctor what their job is. Is it talking to people? Is it writing prescriptions? Maybe it's taking inexact problems from imperfect people with imperfect information, then trying to diagnose and fix or ameliorate problems, within the constraints of cost, time, side effects, and a million other things. Sound familiar?

So how do you measure the productivity of doctors? Given their high cost, obviously the field should be rabid for productivity optimization, right? Doctors have something called RBRVU, or "Resource-Based Relative Value Units.". From that article:

[...] if your organization is measuring physician productivity based on how many patients a doctor sees per day, it needs to take many relativities into consideration. If you compare a primary care physician with a small practice to an ED physician, you are unlikely to see a day when the PCP sees more patients than the bustling ED physician – but is that really a fair and accurate measure of productivity? However, within your organization, if you stack doctors up against those in like-practice, thinking that you can judge productivity on numbers alone, you run into the trap of complexity of care – even within the same speciality, practices may be saddled with patients in varying degrees of medical complexity – and even that will change over time within the same patient!

This seems rather familiar.

Lawyers

Ok, let's try lawyers. Is their job reading briefs? Is it writing them? Is it consulting with people? Or is it doing all of that, while interpreting imperfect laws with imperfect information based on second- and third-hand reports of a situation, while absorbing all of the decisions of the past?

We all are pretty familiar with the traditional method of measuring productivity of lawyers: their billable hour counts. Even there, people are discounting that metric. The only goal of billable hours is higher partner profits. From that article:

The relevant output for an attorney shouldn't be total hours spent on tasks, but rather useful work product that meets client needs. Total elapsed time without regard to the quality of the result reveals nothing about a worker's value. More hours devoted to a task can often lead to the opposite of true productivity. Common sense says that the fourteenth hour of work can't be as valuable as the sixth. Fatigue compromises effectiveness. That's why the Department of Transportation imposes rest periods after interstate truckers' prolonged stints behind the wheel. Logic should dictate that absurdly high billable hours result in compensation penalties.

Hey, there's something interesting. "Useful work product that meets the client needs." How does Scrum define success? Value delivered to the business. It says nothing of how you determine that value. There are too many factors. It may even be impossible to directly correlate revenue to features. Therefore, the only measure of success in scrum is that the product owner is happy.

Developers

So those two fields, often considered where the best and brightest go, have found that hours and other obvious metrics aren't useful to measure productivity. So, why aren't developers treated the same way? Why do we keep being excluded from the "Professional" list?

I'm not suggesting any solution here. I just don't have one. However, it helps explain things like calling developers resources. From that article:

Does George Steinbrenner schedule a “short stop resource” or does he get Derek Jeter?

Do they Yankees want homerun hitting A-Rod or a mere “3rd baseman resource”?

Did the Chicago Bulls staff a “shooting guard resource” or did they need Michael Jordan?

Did Apple do well when it had a CEO “resource” or did they achieve the incredible after Steve Jobs came back to lead the company?

Thoughtworkers and creative types are no different. Software engineers are simultaneously creative and logical, and there is an order of magnitude difference between the best and worst programmers (go read Peopleware if you don’t believe this). Because of this difference, estimates have to change based on the “resource,” which means we’re not interchangeable cogs after all.

You Promised Me Tools!

So let's assume that measuring -- or more importantly, optimizing -- productivity is nearly impossible. How do you keep your team happy and still satisfy the business need for efficient use of capital? Well, what do these other professionals do? Instead of trying to directly measure productivity, they measure anything that impedes productivity.

Measuring Impediments

This is an easy one. Every time something impedes progress, make a note of what it is, and how long it took to resolve. This is especially good to do for any external dependencies. Any time the work leaves the direct, in-progress control of the developer, track who it goes to, and how long they have it.

You can then use this information to talk with the external groups. For example, if the IT folks are taking 2 weeks to turn around a virtual machine, that's a discussion the Dev manager can have with the IT manager. If you have a policy of mandatory code reviews, then track that time. Maybe people are letting those sit around for 3 days, and the manager can set priorities. Maybe there are competing priorities. Either way, the dev manager can show THEIR boss why work items are taking longer than they need to.

Time Before Delivery

This is another interesting metric. Track how long it takes from the point the business requests a work item, to when it's available for use in production. Over time, this metric will stabilize. If the units of work are somewhat consistently sized, predictability will be gained.

Time In Progress

This one tracks the total amount of wall time taken from when work starts on an item, to when it's delivered. Again, if the units of work are approximately similar sized, predictability will be gained here.

Time In Phase

This one tracks the wall time in each phase. Remember how I told you to track external organizations? You should be tracking every phase. The design phase, the dev phase, the QA phase, the code review phase, even the deployment phase. By having every phase tracked, you can identify the slower phases, and see if there is any room for optimization.

Flow Control

Just like working more than 40 hours leads to less productivity, so does working on too much at once. There's a rule of optimization that you can optimize a process only as much as you can optimize a stage. The way to get more done is to remove bottlenecks.

If the QA team is only able to test 4 stories weekly, but developers are finishing 10 stories per week, then only 4 features per week are going to be released. Speeding up the developers will have no effect on the number of features delivered per week. You have to get the QA team to get more throughput. If the managers didn't know the QA team was the bottleneck before, it's impossible to ignore the pile of work that's growing in their phase.

To this end, it makes sense that instead of developers taking a bunch of items on at once, they should focus on one item, and drive it to completion. In addition, there should be some limit of total features being worked on at one time. Work that's being done beyond what the QA team can handle is wasted work. If your developers can help resolve the roadblock in the QA queue, that's going to deliver more value to the customer than working on features. And if we forgot, value is the true output we're trying to deliver.

Wait a Minute...

If you think this all sounds a little familiar, it should. It's the basics of Kanban. It again comes from the manufacturing world, but the focus is on a continuous delivery of value to the customer, with a minimum of wasted work.

We have plenty of articles on the Dev9 blog about Kanban, so I won't go into too much detail here. The basics of Kanban:

Map your value stream. This means separate stages for any handoff point. This also should include any external factors that might impede progress. Then you track the time a story spends in each phase, as described above.

Define the start and end points for the Kanban system. Some teams find if valuable to have To-Do, Doing, and Done. Some teams have Backlog, Design, Dev, Code Review, QA, Release, and Done. It's up to you. Anywhere there's a political or team boundary is a perfect place to have a new phase.

Limit WIP (Work In Progress). As we explained above, increasing productivity of developers without clearing the downstream bottleneck results in wasted work, and no adiditional value delivered to the customer. The team shoul agree on WIP limits, and situations which might allow for breaking those imits.

Service Classes. We know that some production issues will have to take priority. You can have different classes of service (e.g. "standard", "expedite", "fixed delivery date").

Adjust Empirically. Given the data you're tracking above, you can find bottlenecks and inefficiencies, and work to resolve them.

This is the current best solution we've found. Instead of trying to directly measure programmer productivity, which we showed above is practically impossible, focus on measuring anything that impedes their progress, or the progress of delivering value to the customer.

Intuitions

Finally, a little note for you, which is often the antithesis to empirical measurement: trust your gut. Even though you can't just put numbers on it, most developers find it easy to spot good and bad developers. There's just something telling you that they're better. It could be the way they talk about their technology, the thought they put into an answer, or the answer itself. Most developers would sacrifice project and pay to work with a former favorite co-worker. Managers, if you have a developer you like and trust, then trust their input on their coworkers.

In addition, even though they may not be developers, managers often already know who their best and worst performers are. There's usually one or two standout people, even in a team of already-amazing people. If you have all of your developers stack rank each other, it's likely the top performs and the worst performs would be quite consistent. This doesn't fix the issue of finding or hiring developers. The troubles of interviewing could be the subject of an article even longer than this one.

1. https://www.quora.com/How-should-a-software-engineers-productivity-be-measured [↑](#footnote-ref-1)
2. http://emptyeasel.com/2009/05/06/rebecca-elfast-expressive-watercolor-paintings-with-few-brushstrokes/ [↑](#footnote-ref-2)
3. http://wiki.c2.com/?LinesOfCode [↑](#footnote-ref-3)
4. https://redfin.engineering/measure-job-satisfaction-instead-of-software-engineering-productivity-418779ce3451 [↑](#footnote-ref-4)
5. http://engineering.kapost.com/2015/08/you-can-and-should-measure-software-engineering-performance/ [↑](#footnote-ref-5)
6. https://dev9.com/blog-posts/2015/1/the-myth-of-developer-productivity [↑](#footnote-ref-6)