# 9 metrics that can make a difference to today's software development teams

Steven A. Lowe


As I noted in the article "Why metrics don't matter in software development (unless you pair them with business goals)," choosing metrics requires considerable thought and care to support the specific questions a business really needs to answer. Here's the critical point: Measurements should *only* be designed as a way to answer business questions. And those questions are never, "How many KLOCs are we up to now?"

This article picks up where the previous one left off, by discussing first the specific metrics every team should begin using, or at least plan to use soon as a way to improve noticeable performance. Note that the title of this article claims that there are "9 metrics that *CAN* make a difference..."—because what's truly important, what will really make a difference, is how those metrics are actually *used* to improve business value. That use is up to you. Then this article concludes by explaining how you can combine these metrics to create meaning, as well as formulate and test a business value hypothesis.

**World Quality Report 2018-19: The State of QA and Testing**

**GET REPORT**

## Start by measuring the right things

Here are nine objective metrics (marked by bullet points) that you should monitor continuously, to make incremental improvements to processes and production environments. Improvements in these numbers will not guarantee that your customer satisfaction levels will rise by leaps and bounds. But at least these are the right things to measure. In a later section of this article, "Putting it all together," you'll see why.

## Agile process metrics

For agile and lean processes, the basic metrics are leadtime, cycle time, team velocity, and open/close rates. These metrics aid planning and inform decisions about process improvement. While they don't measure success or value added, and they have nothing to do with the objective quality of the software, you should measure them anyway. I'll explain why below.

- **Leadtime**—how long it takes you to go from idea to delivered software. If you want to be more responsive to your customers, work to reduce your leadtime, typically by simplifying decision-making and reducing wait time. Leadtime includes cycle time.

- **Cycle time**—how long it takes you to make a change to your software system and deliver that change into production. Teams using continuous delivery can have cycle times measured in minutes or even seconds instead of months.

- **Team velocity**—how many "units" of software the team typically completes in an iteration (a.k.a. "sprint"). This number should only be used to plan iterations. Comparing team velocities is nonsense, because the metric is based on non-objective estimates. Treating

velocity as a success measure is inappropriate, and making a specific velocity into a goal distorts its value for estimation and planning.

- **Open/close rates**—how many production issues are reported and closed within a specific time period. The general trend matters more than the specific numbers.

When any or all of these metrics are out of the expected range or trending in alarming directions, do not assume a cause. Talk to the team, get the whole story, and let the team decide whether there is cause for concern, and if so, how to fix it.

You cannot know or assume root causes from the numbers, but these metrics give you insight into where your essential processes need attention. A high open rate and a low close rate across a few iterations, for example, may mean that the production issues are currently a lower priority than new features, or perhaps that the team is focused on reducing technical debt to fix entire classes of issues, or that the only person who knew how to fix them quit, or something else. You cannot know or assume root causes from the numbers.

## Production analytics

- **Mean time between failures (MTBF)**
- **Mean time to recover/repair (MTTR)**

Both are overall measures of your software system's performance in its current production environment.

- **Application crash rate**—how many times an application fails divided by how many times it was used. This metric is related to MTBF and MTTR.

Note that none of these three metrics tells you about individual features or users affected. Still, the smaller the numbers, the better. Modern operations-monitoring software makes gathering detailed metrics on

individual programs and transactions fairly easy, but it takes time and careful thought to set up proper ranges for alerts and/or triggers for scaling up or down (for cloud-based systems).

We would like our software to never fail, but that's statistically improbable. When our software does fail, we would like it to never lose any critical data and recover instantly, but that can be extraordinarily difficult to achieve. If your software is your source of revenue, however, the effort is worthwhile.

Beyond MTBF and MTTR, more fine-grained measurements are based on individual transactions, applications, and so on, and they reflect the business value delivered and the cost of remediating failures. If your transaction-processing application crashes one time in a hundred but recovers in a second or two and doesn't lose any critical information, that 1 percent crash rate may be good enough. But if each crash is of an app that runs 100,000 transactions a day, loses a $100 sale, and costs $50 to remediate on the back end, that 1 percent crash rate is going to be a priority. Fixing it will impact the bottom line significantly.

## Security metrics

Security is an aspect of software quality that is often overlooked until later (or too late). Security analysis tools can be used in the build process, in addition to more specialized evaluations and stress tests. Security requirements are often simple and common-sensical, but the software development team needs to be mindful of them, and of the metrics derived from them.

The full range of security practices and related metrics is beyond the scope of this article, but as with agile process metrics and production metrics, there are a few specific metrics that can mean a great deal to your customers' overall satisfaction.

- **Endpoint incidents**—how many endpoints (mobile devices, workstations, etc.) have experienced a virus infection over a given period of time?

- **MTTR (mean time to repair)**—in security metrics, this is the time between discovery of a security breach and a deployed, working remedy. As with the production MTTR metric noted above, the security MTTR should be tracked over specific time intervals. If the MTTR value grows smaller over time, then developers are becoming more effective in understanding security issues such as bugs and how to fix them.

For both these metrics, smaller numbers over time mean you're moving in the right direction. Fewer endpoint incidents speaks for itself. As the MTTR value grows smaller, developers are becoming more effective in understanding security issues such as bugs and how to fix them.

You can find more ways to apply security metrics to software development in the articles Application Security for Agile Projects and Security Threat Models: An Agile Introduction.

## A note on source-code metrics

Today it is easy to plug a source-code scanner into your build pipeline and produce reams of objective metrics. There are empirical averages and suggested ranges and logical arguments about the relative importance of these metrics. But in practice, these tools are most helpful in enforcing coding styles, flagging certain anti-patterns, and identifying outliers and trends.

It's just not worth getting hung up on the numbers. Here's an example, by way of explanation.

Suppose you find a method in a class with a ridiculous metric, such as an NPATH complexity of 52 million. That means that it would take 52 million test cases to fully exercise every path through the code. You could refactor the code into a simpler structure, but before you do that, consider what the business impact would be. Chances are, the old, ugly code works well enough (though its test coverage may be inadequate). Calling out the anti-

pattern to the team so they can avoid repeating it is valuable learning, but fixing it probably wouldn't move the needle on any relevant business metric.

It's best if the team agrees to the level of compliance and rules to which their code is subjected, but be aware that examining outliers and getting concerned about trend blips can waste a lot of time.

## Then put it all together: Success is the ultimate metric

The joy of using automated tools for tracking and measuring quality metrics and user analytics is that it frees up time to focus on the metrics that really matter: success metrics.

## How to use metrics for success

Businesses have goals. Goals imply questions, such as "What does success look like?" or "How will this affect customer behavior?" Properly quantified questions imply metrics.

Put another way, metrics should only be used to answer questions, to test hypotheses that support a specific business goal. And this should be done only as long as the questions and answers help drive positive changes.

Now, don't all projects in general have some set of invariant goals, questions, and hypotheses, and thus metrics?

Yes, but only from the point of view of the business. Business-level measures of things such as user engagement, close rates, revenue generation, and so on provide feedback on how the business is doing in the real world. Changes to software that affect the business will also affect these kinds of metrics.

At a finer level of resolution, every feature and user story may have its own success metric—preferably just one, and one that is directly related to a measure of value delivered to the customer. Completing nine out of ten

stories in a sprint for features that are never delivered is waste, not success. Delivering stories that customers don't want or use is waste, not success. Delivering a story that improves some aspect of user happiness is a success. Delivering a story that demonstrably does *not* improve user engagement is *also* a success, because you will have learned something that didn't work, invalidated a business hypothesis, and freed up resources to pursue other avenues.

## How to formulate a value hypothesis

A value hypothesis is a statement about what you think will happen as a result of the delivery of a specific feature. The relationship between the software, the desired outcome, and the metrics forms the value hypothesis for the feature (or system, or story, or upgrade, etc.). The hypothesis should express how the targeted metric is expected to change, over what time frame, and by how much to be considered effective. You will have to talk to the team and product owner, at minimum, to figure out the specific thing this feature or story is intended to create or improve with respect to the business in order to formulate its value hypothesis. You may have to ask "why" a few times (like a three-year-old) to peel back the layers of unstated assumptions; be patient. When you understand what the business value is supposed to be, you can begin asking the questions that will lead you to the metrics that answer the question.

For example, a "technical" story to improve the speed of an e-commerce checkout process may have as its underlying assumption that a faster checkout will lead to more sales. But why do we think that? Are a lot of people abandoning their shopping carts during the checkout process? If that is the consensus (because that consensus is backed by baseline data), then the value hypothesis may be "We believe that a faster checkout process will result in decreased cart-abandonment rates, leading to higher sales and improved user experience."

You can probably assume that users will like speedier checkout, but it doesn't hurt to ask if they've noticed. Cart-abandonment rates and sales

can be measured before and after the new process is in place, for a period of time. If the cart-abandonment rate drops and sales increase (beyond statistical fluctuations), the evidence supports the hypothesis and you might consider whether even further speed improvements are warranted. If they're not, let this metric fade into the background (or remove it, if it's distracting), and turn your attention to the next hypothesis. If the cart-abandonment rate decreases but sales are unchanged, measure for a longer period of time or rethink the assumed link between cart-abandonment and sales. In other words, **use metrics to learn, and only as long as they prove useful for driving improvements**.

In some cases, the hypothesis may be clearly wrong, so we drop the metrics (and undo the software changes!) after a few days. In other cases, the hypothesis may be correct, so we continue to drive improvements in this area for years.

## Six heuristics for effective use of metrics

We have seen how subjective software metrics matter far more for business success than the traditional, objective quality metrics of old. The effort required to find and measure relevant business metrics for features is outweighed by the insights and learning opportunities gained. Business conditions and opportunities change constantly, so rather than summarize a formula to follow, which may be fragile, here are six rules of thumb, or heuristics, to help maintain focus and flexibility. May they help guide you on your journey to quality software and business success!

1. Metrics cannot tell you the story; only the team can do that (with a hat tip to Todd DeCapula).
2. Comparing snowflakes is waste.
3. You can measure almost anything, but you can't pay attention to everything.
4. Business success metrics drive software improvements, not the other way round.
5. Every feature adds value; either measure it or don't do it.

6. Measure only what matters now.

*Image credit: Flickr*