CHAPTER 2

No Single Metric Captures Productivity

Ciera Jaspan, Google, USA
Caitlin Sadowski, Google, USA

"Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs."

—Bill Gates

"The purpose of software engineering is to control complexity, not to create it."

—Pamela Zave

The urge to measure the productivity of developers is not new. Since it is often the case at organizations that more code needs to be written, many attempts have been made to measure productivity based on lines of code (LOC). For example, in early 1982, the engineering management of developers working on software for the Apple Lisa computer decided to start tracking LOC added by each developer. One week, the main user interface designer, Bill Atkinson, optimized QuickDraw's region calculation machinery and *removed* about 2,000 LOC. The management stopped asking for his LOC [3].

C. Sadowski and T. Zimmermann (eds.), *Rethinking Productivity in Software Engineering*,
https://doi.org/10.1007/978-1-4842-4221-6_2

Although measuring engineer productivity by LOC is clearly fraught, anecdotes like this abound on the Internet [7]. Organizations have continued to search for better and easier ways to measure developer productivity [6]. We argue that there is no metric that adequately captures the full space of developer productivity and that attempting to find one is counterproductive. Instead, we encourage the design of a set of metrics tailored for answering a specific goal.

What's Wrong with Measuring Individual Performers?

Tracking individual performance can create a morale issue, which perversely could bring down overall productivity. Research has shown that developers *do not like* having metrics focused on identifying the productivity of individual engineers [5]; this has also been our experience at Google. Developers are concerned about privacy issues and about how any measurement could be misinterpreted, particularly by managers who do not have technical knowledge about inherent caveats any metric has. If productivity metrics directly feed into an individual's performance grading, then they will impact how developers are compensated and whether they continue to keep their jobs—a serious consequence for getting it wrong. These high stakes further incentivize gaming the metrics, for example, by committing unnecessary code just to increase LOC ratings.

Measuring productivity to identify low performers may not even be necessary. It is our experience that managers (and peers) frequently already know who the low performers are. In that case, metrics serve only to validate a preexisting conception for why an individual is a low performer, and so using them to identify people in the first place is not necessary and serves only to demoralize the higher-performing employees.

Why Do People Want to Measure Developer Productivity?

As critiqued earlier, one possible motivation for measuring developer productivity is identifying high/low-performing individuals and teams. However, there are many reasons why a company may want to measure the productivity of their engineers. Other motivations include surfacing global trends across a company, rating the effectiveness of different tools or practices, running comparisons for an intervention meant to improve productivity, and highlighting inefficiencies where productivity can be improved.

While each of these scenarios has a goal of measuring productivity, the metrics, aggregations, and reporting are different. For example, identifying high- and low-performing individuals means aggregating a metric on an individual level, while running a comparison would mean aggregating across a group of developers. More important, the type of productivity metric used for these scenarios is different. There are many different stakeholders who may be interested in measuring productivity with different goals. If the goal is to identify low performers or to surface global trends, the stakeholders interested in the metric will be looking for metrics that measure task completion. If the goal is to run a comparison for a specific intervention or to highlight inefficiencies within a specific process, the productivity metrics used will be measuring subtasks that address the goals of the intervention or the process being investigated. What is actionable for an individual is different than what is actionable for a team.

What's Inherently Wrong with a Single Productivity Metric?

Any single productivity metric is intrinsically problematic. Productivity is too broad of a concept to be flattened into a single metric, and confounding factors will exacerbate the challenges with attempting such a flattening.

Productivity Is Broad

Productivity is a broad concept with many aspects. The problem is that productivity metrics are poor proxies of the underlying behavior or activity that we want to measure. As poor proxies, they are ripe for misuse.

When we create a metric, we are examining a thin slice of a developer's overall time and output. Developers engage in a variety of other development tasks beyond just writing code, including providing guidance and reviewing code for other developers, designing systems and features, and managing releases and configuration of software systems. Developers also engage in a variety of social tasks such as mentoring or coordination that can have a significant impact on overall team or organization output.

Even for the narrow case of measuring productivity of developers in terms of code contributions, quantifying the size of such contributions misses critical aspects of code such as quality, or maintainability. These aspects are not easy to measure; measuring code readability, quality, understandability, complexity, or maintainability remain open research problems [2, 4].

Flattening/Combining Components of a Single Aspect Is Challenging

Furthermore, flattening all of these into a single measure along with quantity has limited applicability and risks, reducing the actionability of a metric. Is a developer with few code contributions of very high quality more or less productive than a developer with many contributions but some quality issues? Does it make a difference if the engineer with some quality issues comes back and fixes the issues later? It is not clear which is more productive because it depends on the trade-offs of the project in question.

An additional problem with flattening or combining metrics is that flattened metrics may not make intuitive sense and so may be distrusted or misinterpreted. For example, if a variety of factors (e.g., cyclomatic complexity, time to complete, test coverage, size) are compressed into one number representing the productivity impact of a patch, it will not be immediately clear why one patch scores 24 and another one scores 37. Furthermore, a single score is not directly *actionable* since a variety of interrelated factors contribute to that score.

Confounding Factors

Even if we are able to tease out a single metric that holistically covers some aspect of productivity, confounding factors can make the metric meaningless. Take the case of comparing programming languages. It is difficult to measure the productivity of languages in particular because of the number of confounding factors. There is the language itself, the tools, the libraries, the culture, the types of projects, and the types of developers who are attracted to that language.

As another example, a Google team wanted to show that high test coverage improves code quality. To do this, they compared the test coverage of different teams with the number of bugs filed. They found no correlation. Was there really no improvement in code quality, though? In this case, there may have been a confounding *cultural* *component*. Teams that have high test coverage may also file more bug reports. The projects with low test coverage may have been prototypes or just teams that don't track bugs as accurately.

There can also be confounds from *intrinsic complexity differences* between teams. For example, two teams may have a difference in their average patch completion time. One likely explanation is that these teams are working on different projects. There may be project-specific differences in the size of patches they submit or their overall complexity.

There can even be *externalities* that are not captured within a metric. For example, one team might appear to be submitting fewer lines of code than another team. There are many possible causes for such a difference that do not mean the team has lower productivity; perhaps the team is taking more steps to improve quality and therefore has fewer bugs down the road, or perhaps the team has taken on several new employees and is ramping them up. Again, confounding factors are at play. We can't separate those out because they come from nonmeasurable sources.

What Do We Do Instead at Google?

Although there is no general-purpose measurement that can be used in any situation focused on developer productivity, it is still possible to make data-driven improvements to a software engineering *workflow*. Given a specific research question, it is possible to break measurements down into a specific context and know what the caveats are.

At Google, we work with teams to figure out how they can leverage metrics to help make data-driven decisions. The process starts with clarifying the research questions and motivation. We then come up with custom metrics targeted toward those specific questions. This kind of thinking is similar to the Goal–QuestionMetric paradigm [1]. We validate these metrics against qualitative research (encompassing techniques such as surveys and interviews) to ensure that the metrics measure the original goal.

For example, a team at Google working on a distributed version control layer wanted to show that using multiple smaller patches speeds up the review process (perhaps because they are easier to review). After investigating and rejecting not meaningful metrics related to the number of changes or LOC committed per week, the team investigated how long it took developers to commit code *scaled by the size of code changes*. They were able to show improvement in the time to commit per LOC changed.

We can likewise find improvements for other tools, investigate the current cost on developers, and then put those into a Return on Investment (ROI) calculation. For example, we have determined how much time is lost because of waiting for builds (or because of unnecessary context switching as a result of builds). After contrasting this with the cost of speeding up builds (through human or machine resources), we have provided an estimated ROI for different build improvements.

We often see teams that either don't have a research question that matches their motivation for coming up with a metric or have a mismatch between the metrics and the research questions of interest. For example, we talked to one team that wanted to measure codebase modularity. After some discussion, we determined that they wanted to see whether developers were faster at developing software after an intervention and needed to consider ways to measure velocity. Teams also need to carefully consider the time window and aggregations (for example, team versus individual versus larger organization) of interest, as well as any selection criteria for individuals being measured.

Qualitative analysis helps understand what a metric is actually measuring, and data analysis and cross-validation can make sure the results are sensible. For example, by examining distributions of log events for individual developers, we discovered logs that show developers making an action on a web page tens of thousands of times – actions that were actually the result of a Chrome extension. Similarly, we found out during an interview that developers have good reasons for doing something we had thought was an anti-pattern.

Our approach works because we explicitly do not attempt to create a single metric to measure engineering productivity. We instead narrow down the problem into a concrete research statement and seek metrics that address precisely the question at hand. This allows us to validate each individual metric against a specific goal, rather than against the vague concept of productivity. In practice, we find that several of our metrics get reused from one productivity question to the next. While this approach does not scale as fast as applying a single productivity metric, it scales well enough while providing precise, reliable data that we can trust when making investment decisions.

Key Ideas

The following are the key ideas from this chapter:

- There is no single productivity metric for software engineers.
- Instead, focus on a set of custom metrics targeted to a specific question.

References

[1] Basili, V., Caldiera, G., and H. Dieter Rombach. (1994). The goal question metric approach. Encyclopedia of Software Engineering 2, 528–532.

[2] Buse, R. P., & Weimer, W. R. (2010). Learning a metric for code readability. IEEE Transactions on Software Engineering, 36(4), 546–558.

[3] Hertzfeld, A. -2000 Lines Of Code. https://www.folklore.org/StoryView.py?project=Macintosh&story=Negative_2000_Lines_Of_Code.txt

[4] Shin, Y., Meneely, A., Williams, L., & Osborne, J. A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering, 37(6), 772–787.

[5] Treude, C., Figueira Filho, F., & Kulesza, U. (2015). Summarizing and measuring development activity. In Proceedings of Foundations of Software Engineering (FSE), 625–636. ACM.

[6] Thompson, B. Impact: a better way to measure codebase change. https://blog.gitprime.com/impact-a-better-way-to-measure-codebase-change/

[7] Y Combinator. Thread on -2000 LOC Story. https://news.ycombinator.com/item?id=7516671