Chapter 7. Measuring Engineering Productivity

Written by Ciera Jaspen

Edited by Riona Macnamara

Google is a data-driven company. We back up most of our products and design decisions with hard data. The culture of data-driven decision making, using appropriate metrics, has some drawbacks, but overall, relying on data tends to make most decisions objective rather than subjective, which is often a good thing. Collecting and analyzing data on the human side of things, however, has its own challenges. Specifically, within software engineering, Google has found that having a team of specialists focus on engineering productivity itself to be very valuable and important as the company scales and can leverage insights from such a team.

Why Should We Measure Engineering Productivity?

Let's presume that you have a thriving business (e.g., you run an online search engine), and you want to increase your business's scope (enter into the enterprise application market, or the cloud market, or the mobile market). Presumably, to increase the scope of your business, you'll need to also increase the size of your engineering organization. However, as organizations grow in size linearly, communication costs grow quadratically.[1] Adding more people will be necessary to increase the scope of your business, but the communication overhead costs will not scale linearly as you add additional personnel. As a result, you won't be able to scale the scope of your business linearly to the size of your engineering organization.

There is another way to address our scaling problem, though: *we could make each individual more productive*. If we can increase the productivity of

individual engineers in the organization, we can increase the scope of our business without the commensurate increase in communication overhead.

Google has had to grow quickly into new businesses, which has meant learning how to make our engineers more productive. To do this, we needed to understand what makes them productive, identify inefficiencies in our engineering processes, and fix the identified problems. Then, we would repeat the cycle as needed in a continuous improvement loop. By doing this, we would be able to scale our engineering organization with the increased demand on it.

However, this improvement cycle *also* takes human resources. It would not be worthwhile to improve the productivity of your engineering organization by the equivalent of 10 engineers per year if it took 50 engineers per year to understand and fix productivity blockers. *Therefore, our goal is to not only improve software engineering productivity, but to do so efficiently.*

At Google, we addressed these trade-offs by creating a team of researchers dedicated to understanding engineering productivity. Our research team includes people from the software engineering research field and generalist software engineers, but we also include social scientists from a variety of fields, including cognitive psychology and behavioral economics. The addition of people from the social sciences allows us to not only study the software artifacts that engineers produce, but to also understand the human side of software development, including personal motivations, incentive structures, and strategies for managing complex tasks. The goal of the team is to take a data-driven approach to measuring and improving engineering productivity.

In this chapter, we walk through how our research team achieves this goal. This begins with the triage process: there are many parts of software development that we *can* measure, but what *should* we measure? After a project is selected, we walk through how the research team identifies meaningful metrics that will identify the problematic parts of the process. Finally, we look at how Google uses these metrics to track improvements to productivity.

For this chapter, we follow one concrete example posed by the C++ and Java language teams at Google: readability. For most of Google's existence, these teams have managed the readability process at Google. (For more on

readability, see Chapter 3.) The readability process was put in place in the early days of Google, before automatic formatters (Chapter 8) and linters that block submission were commonplace (Chapter 9). The process itself is expensive to run because it requires hundreds of engineers performing readability reviews for other engineers in order to grant readability to them. Some engineers viewed it as an archaic hazing process that no longer held utility, and it was a favorite topic to argue about around the lunch table. The concrete question from the language teams was this: is the time spent on the readability process worthwhile?

Triage: Is It Even Worth Measuring?

Before we decide how to measure the productivity of engineers, we need to know when a metric is even worth measuring. The measurement itself is expensive: it takes people to measure the process, analyze the results, and disseminate them to the rest of the company. Furthermore, the measurement process itself might be onerous and slow down the rest of the engineering organization. Even if it is not slow, tracking progress might change engineers' behavior, possibly in ways that mask the underlying issues. We need to measure and estimate smartly; although we don't want to guess, we shouldn't waste time and resources measuring unnecessarily.

At Google, we've come up with a series of questions to help teams determine whether it's even worth measuring productivity in the first place. We first ask people to describe what they want to measure in the form of a concrete question; we find that the more concrete people can make this question, the more likely they are to derive benefit from the process. When the readability team approached us, its question was simple: are the costs of an engineer going through the readability process worth the benefits they might be deriving for the company?

We then ask them to consider the following aspects of their question:

What result are you expecting, and why?

> Even though we might like to pretend that we are neutral investigators, we are not. We do have preconceived notions about what ought to happen. By acknowledging this at the outset, we can try to address these biases and prevent post hoc explanations of the results.

When this question was posed to the readability team, it noted that it was not sure. People were certain the costs had been worth the benefits at one point in time, but with the advent of autoformatters and static analysis tools, no one was entirely certain. There was a growing belief that the process now served as a hazing ritual. Although it might still provide engineers with benefits (and they had survey data showing that people did claim these benefits), it was not clear whether it was worth the time commitment of the authors or the reviewers of the code.

If the data supports your expected result, what action will be taken?

We ask this because if no action will be taken, there is no point in measuring. Notice that an action might in fact be "maintain the status quo" if there is a planned change that will occur if we didn't have this result.

When asked about this, the answer from the readability team was straightforward: if the benefit was enough to justify the costs of the process, they would link to the research and the data on the FAQ about readability and advertise it to set expectations.

If we get a negative result, will appropriate action be taken?

We ask this question because in many cases, we find that a negative result will not change a decision. There might be other inputs into a decision that would override any negative result. If that is the case, it might not be worth measuring in the first place. This is the question that stops most of the projects that our research team takes on; we learn that the decision makers were interested in knowing the results, but for other reasons, they will not choose to change course.

In the case of readability, however, we had a strong statement of action from the team. It committed that, if our analysis showed that the costs either outweighed the benefit or the benefits were negligible, the team would kill the process. As different programming languages have different levels of maturity in formatters and static analyses, this evaluation would happen on a per-language basis.

Who is going to decide to take action on the result, and when would they do it?

We ask this to ensure that the person requesting the measurement is the one who is empowered to take action (or is doing so directly on their

behalf). Ultimately, the goal of measuring our software process is to help people make business decisions. It's important to understand who that individual is, including what form of data convinces them. Although the best research includes a variety of approaches (everything from structured interviews to statistical analyses of logs), there might be limited time in which to provide decision makers with the data they need. In those cases, it might be best to cater to the decision maker. Do they tend to make decisions by empathizing through the stories that can be retrieved from interviews?[2] Do they trust survey results or logs data? Do they feel comfortable with complex statistical analyses? If the decider doesn't believe the form of the result in principle, there is again no point in measuring the process.

In the case of readability, we had a clear decision maker for each programming language. Two language teams, Java and C++, actively reached out to us for assistance, and the others were waiting to see what happened with those languages first.[3] The decision makers trusted engineers' self-reported experiences for understanding happiness and learning, but the decision makers wanted to see "hard numbers" based on logs data for velocity and code quality. This meant that we needed to include both qualitative and quantitative analysis for these metrics. There was not a hard deadline for this work, but there was an internal conference that would make for a useful time for an announcement if there was going to be a change. That deadline gave us several months in which to complete the work.

By asking these questions, we find that in many cases, measurement is simply not worthwhile…and that's OK! There are many good reasons to not measure the impact of a tool or process on productivity. Here are some examples that we've seen:

You can't afford to change the process/tools right now

There might be time constraints or financial constraints that prevent this. For example, you might determine that if only you switched to a faster build tool, it would save hours of time every week. However, the switchover will mean pausing development while everyone converts over, and there's a major funding deadline approaching such that you cannot afford the interruption. Engineering trade-offs are not evaluated in a vacuum—in a case like this, it's important to realize that the broader context completely justifies delaying action on a result.

Any results will soon be invalidated by other factors

Examples here might include measuring the software process of an organization just before a planned reorganization. Or measuring the amount of technical debt for a deprecated system.

The decision maker has strong opinions, and you are unlikely to be able to provide a large enough body of evidence, of the right type, to change their beliefs.

This comes down to knowing your audience. Even at Google, we sometimes find people who have unwavering beliefs on a topic due to their past experiences. We have found stakeholders who never trust survey data because they do not believe self-reports. We've also found stakeholders who are swayed best by a compelling narrative that was informed by a small number of interviews. And, of course, there are stakeholders who are swayed only by logs analysis. In all cases, we attempt to triangulate on the truth using mixed methods, but if a stakeholder is limited to believing only in methods that are not appropriate for the problem, there is no point in doing the work.

The results will be used only as vanity metrics to support something you were going to do anyway

This is perhaps the most common reason we tell people at Google not to measure a software process. Many times, people have planned a decision for multiple reasons, and improving the software development process is only one benefit of several. For example, the release tool team at Google once requested a measurement to a planned change to the release workflow system. Due to the nature of the change, it was obvious that the change would not be worse than the current state, but they didn't know if it was a minor improvement or a large one. We asked the team: if it turns out to only be a minor improvement, would you spend the resources to implement the feature anyway, even if it didn't look to be worth the investment? The answer was yes! The feature happened to improve productivity, but this was a side effect: it was also more performant and lowered the release tool team's maintenance burden.

The only metrics available are not precise enough to measure the problem and can be confounded by other factors

In some cases, the metrics needed (see the upcoming section on how to identify metrics) are simply unavailable. In these cases, it can be tempting to measure using other metrics that are less precise (lines of code written, for example). However, any results from these metrics will be uninterpretable. If the metric confirms the stakeholders' preexisting beliefs, they might end up proceeding with their plan without consideration that the metric is not an accurate measure. If it does not confirm their beliefs, the imprecision of the metric itself provides an easy explanation, and the stakeholder might, again, proceed with their plan.

When you are successful at measuring your software process, you aren't setting out to prove a hypothesis correct or incorrect; *success means giving a stakeholder the data they need to make a decision*. If that stakeholder won't use the data, the project is always a failure. We should only measure a software process when a concrete decision will be made based on the outcome. For the readability team, there was a clear decision to be made. If the metrics showed the process to be beneficial, they would publicize the result. If not, the process would be abolished. Most important, the readability team had the authority to make this decision.

Selecting Meaningful Metrics with Goals and Signals

After we decide to measure a software process, we need to determine what metrics to use. Clearly, lines of code (LOC) won't do,[4] but how do we actually measure engineering productivity?

At Google, we use the Goals/Signals/Metrics (GSM) framework to guide metrics creation.

- A *goal* is a desired end result. It's phrased in terms of what you want to understand at a high level and should not contain references to specific ways to measure it.
- A signal is how you might know that you've achieved the end result. Signals are things we would *like* to measure, but they might not be measurable themselves.
- A *metric* is proxy for a signal. It is the thing we actually can measure. It might not be the ideal measurement, but it is something that we believe is

close enough.

The GSM framework encourages several desirable properties when creating metrics. First, by creating goals first, then signals, and finally metrics, it prevents the *streetlight effect*. The term comes from the full phrase “looking for your keys under the streetlight”: if you look only where you can see, you might not be looking in the right place. With metrics, this occurs when we use the metrics that we have easily accessible and that are easy to measure, regardless of whether those metrics suit our needs. Instead, GSM forces us to think about which metrics will actually help us achieve our goals, rather than simply what we have readily available.

Second, GSM helps prevent both metrics creep and metrics bias by encouraging us to come up with the appropriate set of metrics, using a principled approach, *in advance* of actually measuring the result. Consider the case in which we select metrics without a principled approach and then the results do not meet our stakeholders’ expectations. At that point, we run the risk that stakeholders will propose that we use different metrics that they believe will produce the desired result. And because we didn’t select based on a principled approach at the start, there’s no reason to say that they’re wrong! Instead, GSM encourages us to select metrics based on their ability to measure the original goals. Stakeholders can easily see that these metrics map to their original goals and agree, in advance, that this is the best set of metrics for measuring the outcomes.

Finally, GSM can show us where we have measurement coverage and where we do not. When we run through the GSM process, we list all our goals and create signals for each one. As we will see in the examples, not all signals are going to be measurable—and that’s OK! With GSM, at least we have identified what is not measurable. By identifying these missing metrics, we can assess whether it is worth creating new metrics or even worth measuring at all.

The important thing is to maintain *traceability*. For each metric, we should be able to trace back to the signal that it is meant to be a proxy for and to the goal it is trying to measure. This ensures that we know which metrics we are measuring and why we are measuring them.

Goals

A goal should be written in terms of a desired property, without reference to any metric. By themselves, these goals are not measurable, but a good set of goals is something that everyone can agree on before proceeding onto signals and then metrics.

To make this work, we need to have identified the correct set of goals to measure in the first place. This would seem straightforward: surely the team knows the goals of their work! However, our research team has found that in many cases, people forget to include all the possible *trade-offs within productivity*, which could lead to mismeasurement.

Taking the readability example, let's assume that the team was so focused on making the readability process fast and easy that it had forgotten the goal about code quality. The team set up tracking measurements for how long it takes to get through the review process and how happy engineers are with the process. One of our teammates proposes the following:

> *I can make your review velocity very fast: just remove code reviews entirely.*

Although this is obviously an extreme example, teams forget core trade-offs all the time when measuring: they become so focused on improving velocity that they forget to measure quality (or vice versa). To combat this, our research team divides productivity into five core components. These five components are in trade-off with one another, and we encourage teams to consider goals in each of these components to ensure that they are not inadvertently improving one while driving others downward. To help people remember all five components, we use the mnemonic "QUANTS":

Qu*ality of the code*

> What is the quality of the code produced? Are the test cases good enough to prevent regressions? How good is an architecture at mitigating risk and changes?

A*ttention from engineers*

> How frequently do engineers reach a state of flow? How much are they distracted by notifications? Does a tool encourage engineers to context switch?

I*n*tellectual complexity

How much cognitive load is required to complete a task? What is the inherent complexity of the problem being solved? Do engineers need to deal with unnecessary complexity?

T*empo and velocity*

How quickly can engineers accomplish their tasks? How fast can they push their releases out? How many tasks do they complete in a given timeframe?

S*atisfaction*

How happy are engineers with their tools? How well does a tool meet engineers' needs? How satisfied are they with their work and their end product? Are engineers feeling burned out?

Going back to the readability example, our research team worked with the readability team to identify several productivity goals of the readability process:

Quality of the code

Engineers write higher-quality code as a result of the readability process; they write more consistent code as a result of the readability process; and they contribute to a culture of code health as a result of the readability process.

Attention from engineers

We did not have any attention goal for readability. This is OK! Not all questions about engineering productivity involve trade-offs in all five areas.

Intellectual complexity

Engineers learn about the Google codebase and best coding practices as a result of the readability process, and they receive mentoring during the readability process.

Tempo and velocity

Engineers complete work tasks faster and more efficiently as a result of the readability process.

Satisfaction

Engineers see the benefit of the readability process and have positive feelings about participating in it.

Signals

A signal is the way in which we will know we've achieved our goal. Not all signals are measurable, but that's acceptable at this stage. There is not a 1:1 relationship between signals and goals. Every goal should have at least one signal, but they might have more. Some goals might also share a signal. Table 7-1 shows some example signals for the goals of the readability process measurement.

Table 7-1. Signals and goals

Goals	Signals
Engineers write higher-quality code as a result of the readability process.	Engineers who have been granted readability judge their code to be of higher quality than engineers who have not been granted readability. The readability process has a positive impact on code quality.
Engineers learn about the Google codebase and best coding practices as a result of the readability process.	Engineers report learning from the readability process.
Engineers receive mentoring during the readability process.	Engineers report positive interactions with experienced Google engineers who serve as reviewers during the readability process.
Engineers complete work tasks faster and more efficiently as a result of the readability process.	Engineers who have been granted readability judge themselves to be more productive than engineers who have not been granted readability. Changes written by engineers who have been granted readability are faster to review than changes written by engineers who have not been granted readability.
Engineers see the benefit of the readability process and have positive feelings about participating in it.	Engineers view the readability process as being worthwhile.

Metrics

Metrics are where we finally determine how we will measure the signal. Metrics are not the signal themselves; they are the measurable proxy of the signal. Because they are a proxy, they might not be a perfect measurement. For this reason, some signals might have multiple metrics as we try to triangulate on the underlying signal.

For example, to measure whether engineers' code is reviewed faster after readability, we might use a combination of both survey data and logs data. Neither of these metrics really provide the underlying truth. (Human perceptions are fallible, and logs metrics might not be measuring the entire picture of the time an engineer spends reviewing a piece of code or can be confounded by factors unknown at the time, like the size or difficulty of a code change.) However, if these metrics show different results, it signals that possibly one of them is incorrect and we need to explore further. If they are the same, we have more confidence that we have reached some kind of truth.

Additionally, some signals might not have any associated metric because the signal might simply be unmeasurable at this time. Consider, for example, measuring code quality. Although academic literature has proposed many proxies for code quality, none of them have truly captured it. For readability, we had a decision of either using a poor proxy and possibly making a decision based on it, or simply acknowledging that this is a point that cannot currently be measured. Ultimately, we decided not to capture this as a quantitative measure, though we did ask engineers to self-rate their code quality.

Following the GSM framework is a great way to clarify the goals for why you are measuring your software process and how it will actually be measured. However, it's still possible that the metrics selected are not telling the complete story because they are not capturing the desired signal. At Google, we use qualitative data to validate our metrics and ensure that they are capturing the intended signal.

Using Data to Validate Metrics

As an example, we once created a metric for measuring each engineer's median build latency; the goal was to capture the "typical experience" of

engineers' build latencies. We then ran an *experience sampling study*. In this style of study, engineers are interrupted in context of doing a task of interest to answer a few questions. After an engineer started a build, we automatically sent them a small survey about their experiences and expectations of build latency. However, in a few cases, the engineers responded that they had not started a build! It turned out that automated tools were starting up builds, but the engineers were not blocked on these results and so it didn't "count" toward their "typical experience." We then adjusted the metric to exclude such builds.[5]

Quantitative metrics are useful because they give you power and scale. You can measure the experience of engineers across the entire company over a large period of time and have confidence in the results. However, they don't provide any context or narrative. Quantitative metrics don't explain why an engineer chose to use an antiquated tool to accomplish their task, or why they took an unusual workflow, or why they circumvented a standard process. Only qualitative studies can provide this information, and only qualitative studies can then provide insight on the next steps to improve a process.

Consider now the signals presented in Table 7-2. What metrics might you create to measure each of those? Some of these signals might be measurable by analyzing tool and code logs. Others are measurable only by directly asking engineers. Still others might not be perfectly measurable—how do we truly measure code quality, for example?

Ultimately, when evaluating the impact of readability on productivity, we ended up with a combination of metrics from three sources. First, we had a survey that was specifically about the readability process. This survey was given to people after they completed the process; this allowed us to get their immediate feedback about the process. This hopefully avoids recall bias,[6] but it does introduce both recency bias[7] and sampling bias.[8] Second, we used a large-scale quarterly survey to track items that were not specifically about readability; instead, they were purely about metrics that we expected readability should affect. Finally, we used fine-grained logs metrics from our developer tools to determine how much time the logs claimed it took engineers to complete specific tasks.[9] Table 7-2 presents the complete list of metrics with their corresponding signals and goals.

Table 7-2. Goals, signals, and metrics

QUANTS	Goal	Signal	Metric
Quality of the code	Engineers write higher-quality code as a result of the readability process.	Engineers who have been granted readability judge their code to be of higher quality than engineers who have not been granted readability.	Quarterly Survey: Proportion of engineers who report being satisfied with the quality of their own code
		The readability process has a positive impact on code quality.	Readability Survey: Proportion of engineers reporting that readability reviews have no impact or negative impact on code quality
			Readability Survey: Proportion of engineers reporting that participating in the readability process has improved code quality for their team

QUANTS	Goal	Signal	Metric
	Engineers write more consistent code as a result of the readability process.	Engineers are given consistent feedback and direction in code reviews by readability reviewers as a part of the readability process.	Readability Survey: Proportion of engineers reporting inconsistency in readability reviewers' comments and readability criteria.
	Engineers contribute to a culture of code health as a result of the readability process.	Engineers who have been granted readability regularly comment on style and/or readability issues in code reviews.	Readability Survey: Proportion of engineers reporting that they regularly comment on style and/or readability issues in code reviews
Attention from engineers	n/a	n/a	n/a
I**n**tellectual	Engineers learn about the Google codebase and best coding practices as a result of the readability process.	Engineers report learning from the readability process.	Readability Survey: Proportion of engineers reporting that they learned about four relevant topics

QUANTS	Goal	Signal	Metric
			Readability Survey: Proportion of engineers reporting that learning or gaining expertise was a strength of the readability process
	Engineers receive mentoring during the readability process.	Engineers report positive interactions with experienced Google engineers who serve as reviewers during the readability process.	Readability Survey: Proportion of engineers reporting that working with readability reviewers was a strength of the readability process

QUANTS	Goal	Signal	Metric
Tempo/velocity	Engineers are more productive as a result of the readability process.	Engineers who have been granted readability judge themselves to be more productive than engineers who have not been granted readability.	Quarterly Survey: Proportion of engineers reporting that they're highly productive
		Engineers report that completing the readability process positively affects their engineering velocity.	Readability Survey: Proportion of engineers reporting that *not* having readability reduces team engineering velocity

QUANTS	Goal	Signal	Metric
		Changelists (CLs) written by engineers who have been granted readability are faster to review than CLs written by engineers who have not been granted readability.	Logs data: Median review time for CLs from authors with readability and without readability
		CLs written by engineers who have been granted readability are easier to shepherd through code review than CLs written by engineers who have not been granted readability.	Logs data: Median shepherding time for CLs from authors with readability and without readability
		CLs written by engineers who have been granted readability are faster to get through code review than	Logs data: Median time to submit for CLs from authors with readability and without readability

QUANTS	Goal	Signal	Metric
		CLs written by engineers who have not been granted readability.	
		The readability process does not have a negative impact on engineering velocity.	Readability Survey: Proportion of engineers reporting that the readability process negatively impacts their velocity
			Readability Survey: Proportion of engineers reporting that readability reviewers responded promptly
			Readability Survey: Proportion of engineers reporting that timeliness of reviews was a strength of the readability process

QUANTS	Goal	Signal	Metric
Satisfaction	Engineers see the benefit of the readability process and have positive feelings about participating in it.	Engineers view the readability process as being an overall positive experience.	Readability Survey: Proportion of engineers reporting that their experience with the readability process was positive overall
		Engineers view the readability process as being worthwhile	Readability Survey: Proportion of engineers reporting that the readability process is worthwhile
			Readability Survey: Proportion of engineers reporting that the quality of readability reviews is a strength of the process
			Readability Survey: Proportion of engineers reporting that

QUANTS	Goal	Signal	Metric
			thoroughness is a strength of the process
		Engineers do not view the readability process as frustrating.	Readability Survey: Proportion of engineers reporting that the readability process is uncertain, unclear, slow, or frustrating
			Quarterly Survey: Proportion of engineers reporting that they're satisfied with their own engineering velocity

Taking Action and Tracking Results

Recall our original goal in this chapter: we want to take action and improve productivity. After performing research on a topic, the team at Google always prepares a list of recommendations for how we can continue to improve. We might suggest new features to a tool, improving latency of a tool, improving documentation, removing obsolete processes, or even changing the incentive structures for the engineers. Ideally, these recommendations are "tool driven": it does no good to tell engineers to change their process or way of thinking if the tools do not support them in doing so. We instead always assume that

engineers will make the appropriate trade-offs if they have the proper data available and the suitable tools at their disposal.

For readability, our study showed that it was overall worthwhile: engineers who had achieved readability were satisfied with the process and felt they learned from it. Our logs showed that they also had their code reviewed faster and submitted it faster, even accounting for no longer needing as many reviewers. Our study also showed places for improvement with the process: engineers identified pain points that would have made the process faster or more pleasant. The language teams took these recommendations and improved the tooling and process to make it faster and to be more transparent so that engineers would have a more pleasant experience.

Conclusion

At Google, we've found that staffing a team of engineering productivity specialists has widespread benefits to software engineering; rather than relying on each team to chart its own course to increase productivity, a centralized team can focus on broad-based solutions to complex problems. Such "human-based" factors are notoriously difficult to measure, and it is important for experts to understand the data being analyzed given that many of the trade-offs involved in changing engineering processes are difficult to measure accurately and often have unintended consequences. Such a team must remain data driven and aim to eliminate subjective bias.

TL;DRs

- Before measuring productivity, ask whether the result is actionable, regardless of whether the result is positive or negative. If you can't do anything with the result, it is likely not worth measuring.
- Select meaningful metrics using the GSM framework. A good metric is a reasonable proxy to the signal you're trying to measure, and it is traceable back to your original goals.
- Select metrics that cover all parts of productivity (QUANTS). By doing this, you ensure that you aren't improving one aspect of productivity (like developer velocity) at the cost of another (like code quality).
- Qualitative metrics are metrics, too! Consider having a survey mechanism for tracking longitudinal metrics about engineers' beliefs. Qualitative

metrics should also align with the quantitative metrics; if they do not, it is likely the quantitative metrics that are incorrect.

- Aim to create recommendations that are built into the developer workflow and incentive structures. Even though it is sometimes necessary to recommend additional training or documentation, change is more likely to occur if it is built into the developer's daily habits.

1 Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering* (New York: Addison-Wesley, 1995).

2 It's worth pointing out here that our industry currently disparages "anecdata," and everyone has a goal of being "data driven." Yet anecdotes continue to exist because they are powerful. An anecdote can provide context and narrative that raw numbers cannot; it can provide a deep explanation that resonates with others because it mirrors personal experience. Although our researchers do not make decisions on anecdotes, we do use and encourage techniques such as structured interviews and case studies to deeply understand phenomena and provide context to quantitative data.

3 Java and C++ have the greatest amount of tooling support. Both have mature formatters and static analysis tools that catch common mistakes. Both are also heavily funded internally. Even though other language teams, like Python, were interested in the results, clearly there was not going to be a benefit for Python to remove readability if we couldn't even show the same benefit for Java or C++.

4 "From there it is only a small step to measuring 'programmer productivity' in terms of 'number of lines of code produced per month.' This is a very costly measuring unit because it encourages the writing of insipid code, but today I am less interested in how foolish a unit it is from even a pure business point of view. My point today is that, if we wish to count lines of code, we should not regard them as 'lines produced' but as 'lines spent': the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger." Edsger Dijkstra, on the cruelty of really teaching computing science, EWD Manuscript 1036.

5 It has routinely been our experience at Google that when the quantitative and qualitative metrics disagree, it was because the quantitative metrics were not capturing the expected result.

6 Recall bias is the bias from memory. People are more likely to recall events that are particularly interesting or frustrating.

7 Recency bias is another form of bias from memory in which people are biased toward their most recent experience. In this case, as they just successfully completed the

process, they might be feeling particularly good about it.

8 Because we asked only those people who completed the process, we aren't capturing the opinions of those who did not complete the process.

9 There is a temptation to use such metrics to evaluate individual engineers, or perhaps even to identify high and low performers. Doing so would be counterproductive, though. If productivity metrics are used for performance reviews, engineers will be quick to game the metrics, and they will no longer be useful for measuring and improving productivity across the organization. The only way to make these measurements work is to let go of the idea of measuring individuals and embrace measuring the aggregate effect.