Editor: **Ciera Jaspan**
Google
ciera@google.com

Editor: **Collin Green**
Google
colling@google.com

A Human-Centered Approach to Developer Productivity

Ciera Jaspan and Collin Green

From the Editors
The "Developer Productivity for Humans" column aims to draw attention to advances and challenges in research and practice in tools and practices that help improve developers' day-to-day tasks. In this column, we reinforce that software engineers and developers are human and productivity tools should support making their jobs easier as opposed to turning them into productivity machines. We share our experiences and expertise and welcome your contributions and feedback.

WE LEAD A mixed-methods research team at Google that seeks to understand what makes engineers productive and happy. We explore the impact of different engineering tools, infrastructure, processes, and best practices on engineering productivity.

Introduction

As part of our job, we regularly meet with and advise Google leaders on what changes they should make (or should not make) to our development tools and processes. These leaders frequently wish to understand—in simple terms—whether productivity is up, down, or stable. They want to know whether their particular tool is making an impact (for example, "Is my framework making developers more productive?"). They hope to see a single metric that clearly goes up or down (and they want "up" and "down" to map unambiguously to "good" and "bad"). Alas, we frequently disappoint them, not because of the estimated effect of their system, but because of the uncertainty around such effects; uncertainty that comes from the fact that measuring developer productivity is inherently difficult.

Why is it so difficult to measure developer productivity?

1. Engineers are humans, and humans are inherently messy.
2. Engineering is a complex and creative task.
3. Measuring the productivity of any knowledge worker is generally a hard problem.

Developer productivity for humans is what our new column is about: how we understand it, how we measure it, and how we improve it. In this article, we'll talk about why this problem is so difficult, and in future installments, we'll get into specific aspects of developer productivity and consider how we might—collectively—improve developer productivity across the industry.

Digital Object Identifier 10.1109/MS.2022.3212165
Date of current version: 23 December 2022

Software Developers Are Humans

Software developers are humans. All of them. It seems that this should be an uncontroversial assertion. Indeed, when we assert it, no one ever disagrees. And yet, we find ourselves making this assertion on a regular basis. Why? Because despite wide agreement with the assertion itself, many people behave as though developers were not humans, but rather nonhuman components in a larger system—cogs in a machine, if you will.

There are two good reasons to think of developers as humans. First, developers strongly prefer that their humanity is acknowledged and respected (a key indicator of their humanity, in fact, is their preference for others to recognize it). Second, if we want to understand what makes developers more or less productive, we need to understand what makes humans more or less productive, with software development as a special case.

There are many things that, in combination, will influence how productive a human will be at a task.

- *The characteristics and limitations of human decision-making processes and memory*: Humans reason in different ways at different times and may select a reasoning approach based on context (for example, time pressure and incentives can push a person into thinking "fast" or "slow"[1]), and the different modes are subject to different strengths and weaknesses.[2] People also have limits to their working memory,[3] so anything we can do to bring the right information to their attention at the right time can improve productivity.
- *The complexity of a task and whether that task is essentially complex or accidentally so*[4]: We can reduce complexity (and therefore increase productivity) by removing accidental complexity.
- *The team of other humans that one needs to work with to accomplish the task*: In addition to the dynamics of the team members, factors such as geographic and time zone dispersion will affect how people communicate, and factors such as a mix of prior experience affect technical mentorship and institutional knowledge.
- *The organizational and business context in which the human completes the task*: The way that projects and people are organized can impact productivity (Conway's law[4,5]). Organizational incentives can also impact productivity: if delivering software fast is rewarded more than delivering software that is reliable, that pressure will shape how work gets done (and what the output looks like). It also shapes the very definition of productivity in that context.
- *The environmental, social, and cultural context in which the human completes the task*: Whether they do it overtly or covertly, individuals bring a whole self to work. They bring their sex, gender, race, ethnicity, nationality, religious background, height, weight, personal beliefs, age, and choices in hairstyle, clothing, and music to work. This introduces a whole other level of complexity to their work and work context. Additionally, the state of the world more broadly (whether it be a global pandemic, geo-political events, press releases from the company, or election results) can change how an individual interacts with others and whether that person might be distracted by outside forces or more sharply focused on the tasks at hand.

These problems are not specific to software development, yet they affect a developer's ability to be productive. Too often, however, people seem desperate to separate "human problems" from "technical problems." There is a persistent belief that "human problems" are tricky and relegated to human resources and psychology departments, while "technical problems" are somehow more tractable. Yet we don't see any difference between the two: "technical problems" frequently require understanding human decision-making processes and performance, and "human problems" can sometimes be addressed by technical solutions. Consider some examples of how they intertwine.

- It's widely accepted that having a faster build speed improves developer productivity. We see evidence that this effect does not happen because a developer sits idly by, waiting for the build to complete. Rather, it occurs because when a build is sufficiently fast, the developer is likely to stay in flow and retain the context of the task. If a build is too slow, a developer will make a very human decision and context switch away to a new task. The developer will also take longer to resume the task when he or she switches back because of the need to regain the context of the task.
- Developers, as humans, are subject to unconscious biases that may affect how they interact

with others through bug reports, design docs, and code reviews. These biases may improve or degrade their own—and others'—productivity and experience. While one could argue that they constitute a "human problem," we can also mitigate such biases with tool changes, such as anonymous code reviews.[6]

So to recap: developers are human, and thus the things that make being a human harder or easier also make being an engineer harder or easier. This isn't to say that there are no factors that specifically affect software developers more than they affect other humans. Flaky tests, build speeds, and technical debt are domain-specific phenomena that have lots to do with software development as an activity. But the influence of domain-specific factors is always accompanied by (and often small in comparison to) the influence of those factors that are general to humans.

Software Engineering Is a Complex, Creative Endeavor

One good reason that we continue to employ humans as software developers, despite all of the messiness described previously, is that humans are good creative problem solvers. Engineering is an inherently creative endeavor in that it involves finding novel solutions to complex problems. Not every solution is novel, and not every problem is complex, of course, but applying known solutions to simple problems doesn't really require "engineering" in any meaningful sense.

A related simplification about engineering work is that it is linear and predictable. We find that engineering leaders, in the face of a complex problem, seek to simplify the entire software engineering process: ideas are generated, designed, implemented, tested, experimented, launched, and maintained. Even engineers are inclined to describe these activities as algorithmic in the most fundamental sense: a prescribed series of steps will inevitably move things from a starting state to a solution.

However, engineering is not a linear or predictable process, and when it is, we consider it "toil" or "boilerplate" and automate away the predictability. This is diametrically opposite of stamping out parts in a machine shop, which is linear and predictable (within some tolerances). So is painting a house, or assembling a rocket, or shoveling coal. Additionally, success in these activities is measurable and unambiguously understood. More coal shoveled is better than less coal shoveled (strictly from a coal-shoveling-productivity point of view). Further, the solution in these examples is about the production of uniform, interchangeable outputs. One pound of coal is pretty much indistinguishable from any other. Not so for code.

- *Software engineering is not algorithmic*: There is no prescribed set of steps that will take one from having no functioning code to having functioning code. While there are general processes or best practices (agile methods, test-driven development, modular designs, fault-tolerant architectures), engineers adapt to the current problem and take alternative paths as necessary.
- *The output of software development is not known from the start*: In part, that's because the solution is not known ahead of time. It may be well constrained, and it may look a lot like solutions that have been built in the past, but even where code reuse is extensive and a developer draws on bits and pieces from others' solutions, the output of a software development task is unique.
- *Software development is not about the production of uniform, interchangeable outputs*: Given that the solution isn't known at the outset and given that the eventual solution is unique, it should be no surprise that the products of software development are neither uniform nor interchangeable. Not all programs are equal, not all files or functions are equal, and not all lines of code are equal.

Similar to the assertion that developers are human, the assertion that software engineering work is nonlinear and unpredictable is often met with agreement. But again, we see attempts to simplify engineering work to make the problem of engineering productivity more tractable. This simplification leads to failures to treat engineering as appropriately complex and creative in practice. For example:

- *Conflating throughput with productivity*: One might count the lines of code written by a developer in a certain period and calculate a simple output per unit time "productivity" measure like lines of code per minute. Pounds of coal shoveled per hour will tell you which shovelers are the best shovelers; lines of code per minute will not tell you which software developers are the best software developers.
- *Assuming the built product is the right product*: Some code

that gets submitted and deployed is terrible. It might have performance issues or introduce bugs. It might be brittle or scale poorly. It might be difficult to comprehend, modify, or maintain. It might even work perfectly, yet be the wrong market fit. Part of productivity is not just the output but whether it was the right output to build.

- *Assuming work that doesn't result in output has no value*: Software developers do a lot of cognitive work. They think through a problem, they look for analogous solutions, and they learn what tools, libraries, and technologies are at their disposal and how to use them. Problem solving (because that's what this really is) involves a bunch of work that doesn't result in an immediate objective output, is hard to measure, and may pay long-term dividends in productivity. Maybe.

To recap: software engineering is complex and creative. It is problem solving at its core. It's pretty much nothing like shoveling coal, and any attempt to treat it similarly in hopes of understanding developer productivity is going to miss the mark.

Measuring Productivity Is Hard

We've made a few not-very-controversial assertions so far: that developers are human, that humans are messy, that software development involves creative problem solving, and that humans are good at software development because they're good at problem solving. Given that none of these broader observations is really controversial and that humans have been trying to measure productivity systematically for at least a century, haven't we progressed further with measuring developer productivity? Yes. And no.

For starters, modern attempts to quantify and analyze work productivity began with Frederick Taylor,[7] who was, in fact, measuring productivity for tasks like shoveling coal, moving heavy objects, and operating machinery in known, repeatable patterns. Taylor held four principles of scientific management, which might be paraphrased:

1. Don't make guesses about efficiency and productivity; measure and evaluate them systematically.
2. Select, train, and cultivate workers deliberately.
3. Decompose work into tasks that can be delegated to workers (ideally, along organizational lines).
4. Provide specific, prescriptive task instructions to each worker and monitor them to ensure they execute as directed.

Taylor's approach feels rigorous and objective and dispassionately scientific. But does it apply to developer productivity? We have no complaint about the first principle; we systematically study developer productivity, and we think it's a good idea. Similarly, the second principle doesn't present a problem: thoughtful hiring, deliberate training and mentoring, and a focus on developing and retaining developers are table stakes (though overly rigid one-size-fits-all notions of how to do those things are problematic).

The third principle presents some problems. The act of decomposition of work into tasks is itself a software engineering design, process, and management problem. There are entire books dedicated to the decomposition and modularization of code such that tasks can be more easily delegated across a team with lower communication overhead. This decomposition, though, is itself an engineering task and is arguably much more difficult and time consuming than the completion of the decomposed work.

The fourth principle cannot be implemented without the third, but even if it could, it presents a problem: pretty much no human likes being surveilled in this manner. They like it even less if their work is such that outward indications of productivity are not always apparent when, in fact, progress is being made via thinking, learning, or experimentation.

At the beginning of this article, we mentioned that we study what makes engineers productive and happy. Productive engineers might be unhappy and—despite feeling productive—decide to go someplace else for a job. They're human after all. Despite systematic selection processes, developers are not interchangeable. When a senior, long-tenured engineer leaves an organization, it is impossible to simply drop in a replacement engineer who has been in cold storage. Attrition (whether on good or bad terms) has a cost in productivity and resources, and when productivity measurements (especially myopic and inappropriate measurements, like mere throughput) are foisted upon developers, they are likely to become unhappy.

So the original flavor of scientific management isn't suited to measuring developer productivity (or, really, any kind of knowledge worker productivity); this is a point that others have made before. Management science has evolved in its methods and philosophy (see Drucker[8] and Ebert and Freibichler[9]). Drucker noted that: "Knowledge-worker productivity is the biggest of the 21st century management challenges" (Drucker, p. 157).[8] Despite taking a more complete and nuanced view of management science, these more current works continue to struggle with the question of how to measure productivity, specifically. Drucker does acknowledge the need to focus on quality of outputs over their quantity, and he embraces the idea that productivity is complex and involves tradeoffs. However, these authors focus on discarding the idea that one can implement Taylor's principles for knowledge work (we agree on this) and talk about other ways of trying to improve productivity, but they fall short of suggesting a measurement strategy.

Developer Productivity for Humans

So what can one do, given the mushy mess that is measuring productivity for a bunch of humans doing a complex, creative thing?

We need to think about measuring productivity in a holistic and multifaceted way, not in a reductionist, unidimensional way.[10] Accordingly, we need to measure productivity using more than one metric, and we need frameworks for selecting metrics (for example, SPACE from Forsgren et al.)[11] that enable us to understand tradeoffs. We must think about productivity both in the short term and the long term; for example, we need to understand the effects that biases against underrepresented groups have on developer productivity in terms of getting code submitted and also on retaining skilled employees by treating them fairly and affording them the same opportunities that others enjoy.

ABOUT THE AUTHORS

CIERA JASPAN the software engineering lead for the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact her at https://research.google/people/CieraJaspan/ or ciera@google.com.

COLLIN GREEN is the user experience research lead for the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact him at https://research.google/people/107023/ or colling@google.com.

We also need to remind our stakeholders that developers are human. Well, not remind perhaps—it's not something that's forgotten so much as overlooked. We must keep the fact that developers are human in focus as we create metrics and measurement strategies. It's critical that developer productivity metrics are human centered. This makes the problem harder (as we've discussed), but it's also the only way to do the problem justice and make real progress.

In future installments of this column, we'll talk more about how we're trying to do all of this at Google. For each article, we'll explore one problem within developer productivity, and we'll take a holistic, human-oriented view toward understanding the problem space, how to measure it, and how to improve it. We'll draw on our own team's research but also on the amazing research done by colleagues at other companies and across academia. We'll cover a wide range of topics, as diverse as the future of hybrid and distributed teams, flaky tests, inclusive teams, code quality, ramping up new hires, and technical debt. Each of these subjects involves both human and technical considerations, each of them is a complex topic, and each is very tricky to measure. Yet we can use the same holistic, human-oriented way to understand these topics and make real improvements to developer productivity. We hope you enjoy exploring these topics with us, and we look forward to hearing from our readership!

References

1. D. Kahneman, *Thinking, Fast and Slow*. New York, NY, USA: Macmillan, 2011.
2. A. Tversky and D. Kahneman, "Judgment under uncertainty: Heuristics and biases," *Science*, vol. 185, no. 4157, pp. 1124–1131, 1974, doi: 10.1126/science.185.4157.1124.
3. A. Baddeley, "Working memory," *Science*, vol. 255, no. 5044, pp. 556–559, 1992, doi: 10.1126/science.1736359.

4. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA, USA: Addison-Wesley, 1975.
5. M. Conway, “How do committees invent?” *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.
6. E. Murphy-Hill, C. Jaspan, C. Egelman, and L. Cheng, “The pushback effects of race, ethnicity, gender, and age in code review,” *Commun. ACM*, vol. 65, no. 3, pp. 52–57, 2022, doi: 10.1145/3474097.
7. F. W. Taylor, *The Principles of Scientific Management*. New York, NY, USA: Harper & Brothers, 1911.
8. P. F. Drucker, *Management Challenges for the 21st Century*. Oxford, U.K.: Butterworth-Heinemann, 1999.
9. P. Ebert and W. Freibichler, “Nudge management: Applying behavioural science to increase knowledge worker productivity,” *J. Org. Des.*, vol. 6, no. 1, 2017, Art. no. 4, doi: 10.1186/s41469-017-0014-1.
10. C. Jaspan and C. Sadowski, “No single metric captures productivity,” in *Rethinking Productivity Software Engineering*, C. Sadowski and T. Zimmermann, Eds. Berkeley, CA, USA: Apress, 2019, pp. 13–20.
11. N. Forsgren, M. A. Storey, C. Maddila, T. Zimmerman, B. Houck, and J. Butler, “The SPACE of developer productivity: There’s more to it than you think,” *ACM Queue*, vol. 19, no. 1, pp. 20–48, 2021, doi: 10.1145/3454122.3454124.
12. S. B. Johnson, *The Secret of Apollo: Systems Management in American and European Space Programs*. Johns Hopkins Univ. Press, 2002.

Digital Object Identifier 10.1109/MS.2022.3223243