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

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

Developer Productivity for Humans, Part 4: Build Latency, Predictability, and Developer Productivity

Ciera Jaspan and Collin Green

THIS WEEK, WE'RE going to tackle what was once the most common complaint among developers at Google: "My build is too slow." At first glance, this does not seem like a human-centric topic. After all, build latency itself is a purely technical problem: we can solve it by making faster compilers, reducing dependencies, and using incremental compilation. We can measure the reduced latency directly, but is it safe to assume that reduced latency translates directly to saved time or increased productivity? Probably not without some caveats: Improvements that reduce build latency translate to overall time savings or increased developer productivity inasmuch as they affect the developer as a human. The developer may or may not notice the reduced latency and update their expectations about builds accordingly. The developer may or may not make different choices about how to structure their work based on those expectations about how fast they can get information from their build system. Reducing build latency is a technical problem, but fully understanding the benefits of doing so involves understanding developers as humans.

Digital Object Identifier 10.1109/MS.2023.3275268
Date of current version: 14 July 2023

Everyone's Favorite Complaint: Build Latency

In this article, we'll do a deep dive into build latency. How fast do builds need to be for developers to stay productive? Are there changes we can make to build systems besides just "make builds faster" to improve productivity? And how much of a productivity improvement can we reasonably expect by improving build latency, anyway?

Looking for the Magic Number

When we first proposed to work on build latency, our leadership had a very simple question for us: "How fast do builds need to be for developers to stay on task and be productive? Where's the 'knee' in the graph of build latency by productivity"?

This question presupposes a particular state of the world. It assumes that as build latency increases, there is some threshold at which developers are more likely to go off task or otherwise be unproductive. In theory, the relationship looks something like Figure 1.

Our first task therefore was to try to find the magic productivity threshold. To do this, we looked at our developer logs to understand the following:

1. During the course of the build, how long does it take before developers go "off task" to work

on another project, check email, or just stop working on the current task?
2. When a build completes, how fast do developers return to their task and resume making progress?

We could do this because we were able to track developer actions across their tools, as described.[1] These actions were also associated with artifacts such as the change the developer is working on, the file being edited, the documentation being viewed, etc. As the build was also associated with a change, we were able to determine when the developer had switched to a different task and when they had returned to their original task.

The expectation was that there would be a clear pattern where, if the build takes less than x seconds, it would mean developers are more likely to stay on task and more likely to return quickly to their task. Reality has a nasty habit of dashing our expectations, though. We found ourselves looking for the "knee" in real data that looked more like Figure 2.

There is no knee, no magic number. Every improvement to build latency will help developers stay on task, and get back on task, faster. While it's disappointing to not have an ideal target number, it's also an opportunity. Every change to build latency can increase the likelihood of developers staying on task, although if there are longer build latencies, one would need a proportionally larger change to see an impact.

This isn't the whole story, of course. As builds get faster, incremental improvements are harder to achieve. From a resource investment perspective, there's a point at which it becomes impractical to further reduce build latency. However, looking only at the relationship between latency and task switching, faster builds are always beneficial.

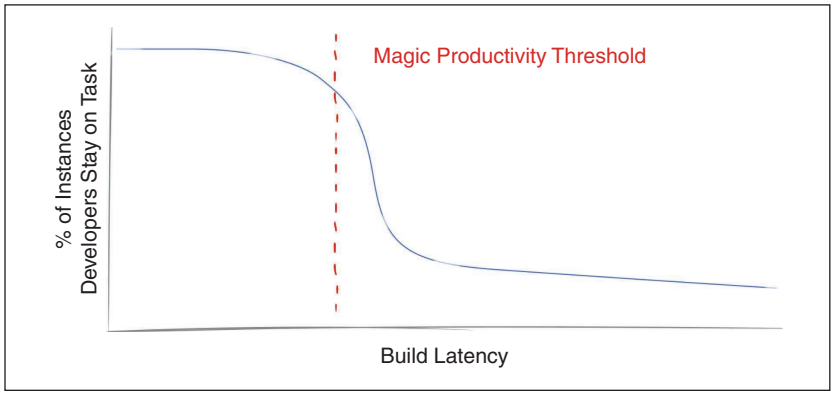

FIGURE 1. The hypothesized relationship between build latency and the likelihood of developers staying on task.

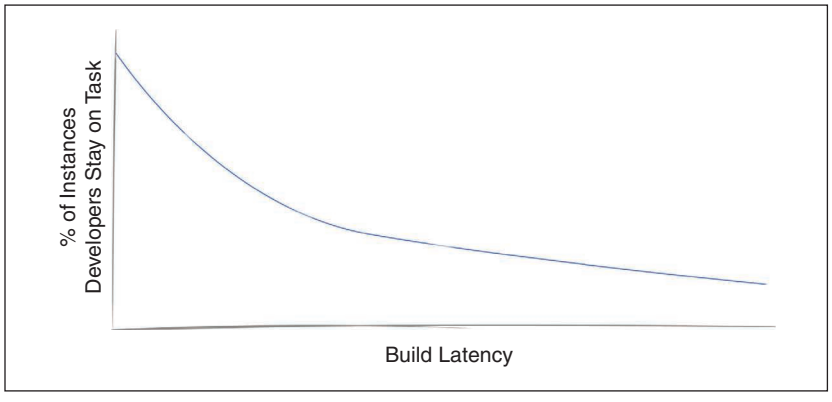

FIGURE 2. The actual shape of the relationship between build latency and the likelihood of developers staying on task.

Humans Aren't Great at Time Estimation

Why isn't there a magic productivity threshold? Why don't developers stay on task for short builds and task switch for long builds in a systematic manner? We investigated these questions and found that developers often don't know how long a build will take, and so they cannot optimize for build latency. Based on our research, we hypothesize that build systems can provide developers with better latency estimates to help them determine when to context switch.

To understand how developers think about tasks, workflow, and their builds, we ran an experience sampling study over the course of two weeks. During the study, every time a developer started a build, we sent them a chat message with a very short survey. We asked the developers the following:

1. How long they expected their build to take. We provided time buckets: "under 10 seconds," "under 1 minute," "1–2 minutes," "2–5 minutes," "5–10 minutes," "10–20 minutes," "20–30 minutes,"

"30–60 minutes," and "over 60 minutes."
2. What they had been doing since starting the build. We provided a list of common activities, including "working on this task," "working on another task," "checking email," and "non-work activity," and we included a write-in option as well.

We learned from this that developers choose what to work on based on how much time they *think* they will have, not how much time they *actually* have. This seems obvious in retrospect. If a developer thinks the build will take over 60 min, they might go get lunch. If they think it will take a few minutes, they might go do a short code review. If the developer thinks the build will take under 10 s, they inspect their code and stay focused on their task.

The problem is that developers were *quite inaccurate* at estimating build latency for individual builds. In our study, developers selected the wrong bucket 65% of the time. And regardless of whether the developer overestimates or underestimates, they're going to be negatively affected by their build latency.

- Consider a developer who overestimates build time: The developer thinks that the build will take 5 min, but it actually takes 30 s. Because the developer thinks the build will take 5 min, they walk off to go get a cup of coffee. They come back, and the build is complete. The developer's flow was broken: they not only delayed the progress of their task, but they'll likely pay a small penalty associated with the cognitive overhead of task resumption.
- Consider a developer who underestimates build time: They think the build should take about 30 s, so they wait for it to finish. However, it takes 5 min. The developer is now annoyed and perhaps could have responded to an email or updated a bug instead of waiting.

Either way, we have a developer who is going to feel that build latency is disrupting their workflow and harming their productivity. And they are right: it is! However, the harm comes not necessarily from the build latency itself but from its unpredictability and the resulting inability to task switch (or not) effectively. If build latencies were consistent *or* if the build system could usefully inform the developer of expected latency, the developer could make a better decision about what to do next. Reducing build latency is not our only lever to improve productivity here; we can also improve developers' ability to make decisions around build latency as it is.

We also explored how consistency of build latency relates to developers' satisfaction with build latency. Using our quarterly survey, we asked developers about their satisfaction with build latency and plotted the results of each developer's standard deviation of the builds they had run in that quarter (Figure 3). We found that the developers who were the most dissatisfied with build latency were also the ones that had the largest standard deviations. Interestingly, the effect was substantially smaller when we plotted developers' average build latencies, which again indicates that *predictability* is possibly even more important than actual speed. A predictable event allows the developer to better plan their work to increase their own efficiency.

Of course, to act on this insight, one needs to separately consider whether build systems can reliably predict build latency with adequate performance to inform the developer in the moment of decision making about what to do next. We haven't taken this step at Google, but we think that the idea holds promise for improving developer productivity and satisfaction.

Even Modest Build Latency Improvements are Helpful

To recap, we see evidence that faster builds are better, both because they require less time and because developers will task switch less often when builds are quick. Importantly, we

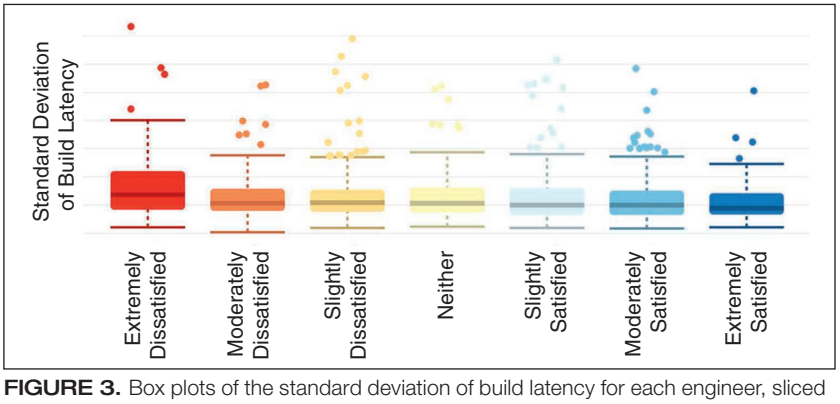

FIGURE 3. Box plots of the standard deviation of build latency for each engineer, sliced by developer satisfaction with build latency. The boxes show the interquartile range.

see evidence that developers will be much better at optimizing their task switching when builds are predictable (or predicted for them), regardless of their speed. What is the real-world benefit of build latency reduction?

The question of whether to pursue infrastructure improvements that will speed up builds is ultimately a consideration of costs and benefits. The *costs* of such upgrades are often straightforward to calculate, but the *benefits* can be harder to quantify, in part because of uncertainty in how developers will respond to incrementally faster builds (as above, will they notice and—if so—will they change their behavior in a productive manner?).

A few years ago, the team responsible for our build machines ran into exactly this problem. They had a new system, which was more expensive to run, but it would deliver faster build latencies. The problem was that the expected improvement was very modest: their pilot showed a 15% improvement in build latencies overall. If it was a 50% improvement, it would have been a more obvious win: taking a 1-min build to 30 s, or a 20-s build to 10 s, would almost certainly have a productivity impact. However, 15% is less clearly going to help: if your 1-min build is now 51 s… does it even matter? Do you notice? The team asked us for assistance in evaluating whether the more expensive system was worthwhile.

To evaluate this, we performed a blind experiment: 15% of developers were selected to have their builds supported by upgraded machines (this is the experiment group), while 85% of developers had their builds supported by existing (not upgraded) machines (the control group). Those developers assigned to the experiment group experienced faster builds from the upgraded machines but not drastically so. During the study, the median developer in the experiment group saw builds improve by just a few seconds on average (a 13% reduction in median build time per developer).

We organized data collection to evaluate several outcome measures:

- Self-reported productivity, self-reported velocity, and satisfaction with build latency as measured through our quarterly surveys. The developers were not aware that we were running an experiment; this was part of our regular survey cadence.
- The number of times developers ran a build each week and the number of lines of code they submitted, measured through logs.
- The median wall-clock coding time and median active coding time on the change lists the developer submitted. The wall-clock time is the time from the first edit to the change list being submitted, while the active time is the "fingers on keyboard" time, including time spent in the integrated development environment and also time spent looking up information. (See Jaspan et al.[1] for information on how we extract these metrics from log data.)

We then ran the experiment for three months. Due to an unexpected change in the third month (see below), we extended it for an additional two months. We analyzed the data using a difference in differences method with an individual fixed-effect model to control confounding factors specific to the developer.

Despite not knowing they were part of the study, our experiment group showed slightly higher self-reported productivity (a four-percentage-point increase to the percentage of developers reporting they were at least moderately productive) and slightly higher self-reported velocity (a five-percentage-point increase in those who reported being satisfied with their velocity). We also found that satisfaction with build latency increased by five percentage points in the experiment group. All increases were statistically significant increases over the control group. While these are modest improvements, they are surprisingly large given the relatively small change to build latency.

The most surprising result of the study was from the behavioral metrics: the number of builds they did a week, the active coding time and wall-clock coding time it took developers to create each change list, and the number of lines of code they produced. For the first two months of the study, we saw no changes to the experiment group or the control group in these metrics. In the third month, though, we saw a slight improvement in these metrics in the experiment group: that group, on average, ran one more build a week and submitted 24 more lines of code per week. Additionally, the developers in the experiment group were faster to complete small- to medium-length change lists (11% faster active time and 14% faster wall-clock time). This delayed change in behavior was a surprise, so we extended the experiment for two additional months. The behavior was sustained.

What happened here? Our best hypothesis is that despite the very modest change to build latency, the developers in the experiment group *adapted* to the faster build latencies. They were able to fit in one more build a week, which meant just a few more lines of code submitted each week. Their slightly faster

iteration time resulted in overall velocity improvements for smaller changes, as well.

In summary, an incremental change in build latency has several effects on developers' behavior and perception (albeit at a delay) that can and should be considered benefits of faster builds. Admittedly, these behavioral benefits are hard to estimate and are unlikely to change in a linear or monotonic manner with build latency increases.

Build latency reductions are important for developer productivity, but these improvements are filtered through a lens of human perception and judgment. As builds get longer, developers are more likely to task switch (which itself has productivity consequences), but there's not a magic number that will ensure developers stay on task. Additionally, it's not just absolute build latency that's important: the developer needs to be able to accurately predict build latency to get the best productivity gains and optimize their day. Coffee breaks have to go in somewhere; it's best if they can overlap with a longer build.[2]

Through experimentation, we've confirmed that even moderate improvements to build latency result in changes to developer behavior that indicate greater productivity: more builds, more lines of code written, and faster completion times for small/medium changes. However, again, there is human judgment in the loop here: the developer has to (explicitly or implicitly) notice the change, integrate it into their expectations, and adapt their day to their new working model. In practice, we observed that it took two months for developers to adapt for a moderate change to latency.

Even if you can't actually improve build latency, though, you can improve the *predictability* of build latency, either by making builds take similar lengths of time or by informing developers of how long (approximately) they are expected to take. None of this should actually surprise us. Developers are human, and we've previously discussed that this is an important factor in understanding developer productivity.[3] One current view in psychology is that human behavior is best understood as rational behavior (i.e., optimization) within the constraints of the environment, current goals and tasks, and human cognitive, perceptual, and motor constraints.[4] Developers are working in an uncertain environment. They're doing complex tasks that are hierarchical and interrelated. They're doing these tasks (of course) within the bounds of their own human performance. Build latency is one narrow example, but a general approach to improving productivity falls out of this discussion: if you can't make a process faster or easier, at least make it more predictable so that developers can optimize around it.

ABOUT THE AUTHORS

CIERA JASPAN is 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.

References

1. C. Jaspan et al., "Enabling the study of software development behavior with cross-tool logs," *IEEE Softw.*, vol. 37, no. 6, pp. 44–51, Nov./Dec. 2020, doi: 10.1109/MS.2020.3014573.
2. R. Munroe. "Compiling." xkcd. Accessed: May 23, 2023. [Online]. Available: https://xkcd.com/303/
3. C. Jaspan and C. Green, "A human-centered approach to developer productivity," *IEEE Softw.*, vol. 40, no. 1, pp. 23–28, Jan./Feb. 2023, doi: 10.1109/MS.2022.3212165.
4. F. Lieder and T. Griffiths, "Resource-rational analysis: Understanding human cognition as the optimal use of limited computational resources," *Behavioral Brain Sci.*, vol. 43, p. e1, Mar. 2020, doi: 10.1017/S0140525X1900061X.