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

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

What Happens When Technical Debt Vanishes?

Ciera Jaspan and Collin Green

HERE'S A THOUGHT experiment. Say we wave a magic wand across a codebase and an entire class of technical debt, poof, goes away and immediately evaporates if introduced in the future. For example, maybe we make it so that dead feature flags just delete themselves as soon as the engineer wills it; maybe large-scale migrations just migrate themselves; maybe we magically have 100% test coverage, without an engineer lifting a finger. (How does this magic work? Perhaps problems are prevented with presubmit checks. Perhaps static analysis. Perhaps it's AI. The specific magic doesn't matter; it only matters that technical debt goes away.)

What will happen to developer productivity? Surely, it increases.

But will the metrics that we use as proxies for "developer productivity" improve?

What is Technical Debt Again?

Financial debt arises when one decides to borrow money now that will be paid back—with interest—later. Technical debt arises when one allocates engineering capacity now (to achieve some goal) in such a way that the spent capacity will have to be paid back—along with some overhead (in the form of rework or interim maintenance or friction)—later.

> "In software-intensive systems, Technical Debt is a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible."[1]

When a team deliberately and prudently[a] takes on technical debt, they are trading off the realization of business value *now* for having to pay off some technical debt *later*.[2] Leveraging the financial metaphor, we can talk about the difference between paying off the *principal* and the *interest*.[3]

- When we pay off the principal, we *eliminate* the accrued technical debt (e.g., completing a major migration). Paying off the principal also implies avoiding future interest payments.
- When we pay (only) the interest, we deal with the consequences of having technical debt in our codebase (e.g., handling defects that would have been prevented by missing tests). Paying the interest involves the ongoing effort required to work around technical debt as well as any increased cost of eliminating it that results from having built around it over time.

Of course, allocating engineering capacity is not quite as simple as taking a loan. When a team incurs technical debt, they estimate the principal, the interest, and the immediate business value of doing so. It's like taking out a loan, but estimating the size of the loan, interest rate, duration of the loan, and the business value achieved from taking it on at all. Taking on technical debt is decision making under uncertainty.

Accepting such uncertain terms sounds scary, but the reality is that there is no alternative. *Every* investment of engineering capacity involves uncertainty and estimates of value. Aside from sitting idly and doing nothing, trying to invest capacity wisely despite uncertainty is the only option. That's why engineering organizations incur technical debt (and probably always will).

Could Technical Debt Vanish?

This idea of eliminating an entire category of technical debt might

[a] https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Digital Object Identifier 10.1109/MS.2025.3621709
Date of current version: 23 December 2025

seem absurd, but our field has done this before. Consider code autoformatters: In the days before autoformatters, lint errors were considered a form of technical debt worth measuring and fixing. There existed tools to count and catalog them. Today, most lint errors are addressed through autoformatters; engineers don't waste time formatting code or fixing old code that's not formatted because it's trivial to clean up as needed.

Think bigger! Formatting errors are not our real problem. What if we could eliminate a larger category of technical debt? At Google, engineers' most-reported forms of technical debt are migrations that are needed or in progress, poor test quality or coverage, and hard to find, incomplete, or missing documentation. What if we reduce or eliminate the cost to pay this debt back?

Consequences of Eliminating Technical Debt

What We Might See in the Engineering Organization

One consequence of having a technical debt magic wand is that accomplishing any specific goal is lower cost. If the magic wand does work that engineers would otherwise do, we've reduced the engineering hours required to meet that goal. For example, if our magic wand eliminated the problem of missing or inadequate tests, then completing our software project while meeting a prescribed bar for test coverage and quality would require less test-writing time.

Saving engineering hours would likely *not* equate to keeping fewer engineers on staff.

Engineering hours are a resource, and our new magic enables us to use them more efficiently, so there's a good chance that engineering organizations would try to do more with the same population of engineers or even seek to grow their staff, a realization of Jevon's Paradox[b] that is already being discussed by economists.[4] Engineers' time would likely be reallocated to higher-value work. Some of that work would be creation of novel features or functionality ("greenfield development"), and some of it would be maintenance or improvement of existing products (i.e., eliminating technical debt that we weren't able to magic away or "brownfield development"). Inevitably, some of this work would incur new technical debt.

A likely outcome is that an engineering organization would—after a period of adaptation—settle into a mode of operation that incurred and repaid technical debt in proportion to the relative risk appetite and tolerance[5] of the organization. The flavor of technical debt that can be magically eliminated would be incurred more readily than before and would cost nothing to eliminate. The flavors of technical debt that still require traditional repayment might be incurred faster (with more developers doing greenfield development) and repaid more readily (with more developers doing maintenance and improvement).

This outcome is speculation, but it is consistent with how we understand engineering and business organizations to think about technical debt and decision-making.

[b]https://en.wikipedia.org/wiki/Jevons_paradox

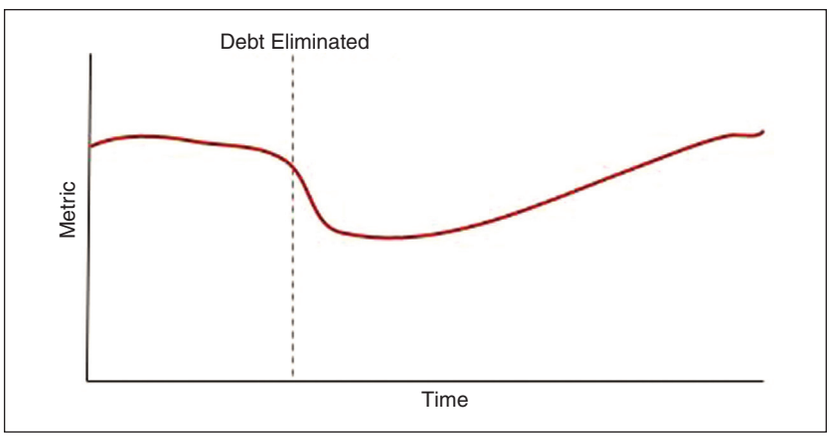


FIGURE 1. Some metrics (e.g., reported hindrance from technical debt) might rapidly improve but return to their baseline value over time.

What We Might See in Productivity Metrics

If our speculation about what would happen in the engineering organization is correct, how might that be reflected in productivity measurements? Initially, we would expect engineers to report less technical debt in surveys. After all, if we eliminate a bunch of technical debt, engineers will run into it less frequently.

Would that pattern persist? One hypothesis is that the number of engineers reporting being hindered by technical debt will drop temporarily, and then rise again (see Figure 1)

With some forms of technical debt removed, engineers will take on *different kinds of technical debt* as part of the work they do instead. As described previously, their time will be freed to do new work and

that new work will incur new technical debt. This is a good thing: Each development decision is a tradeoff between realizing new value and paying back technical debt.

It is unlikely that all subjective measurements would show such rebound effects: On surveys we ask engineers about the extent to which specific kinds of technical debt impede their work (e.g., "How much have missing or inadequate tests hindered your productivity?") and the introduction of a magic wand would durably reduce engineers' reports of the technical debt waved away with the magic wand.

What about an objective metric? What if we had a "maintenance overhead" metric[6] that captures the proportion of time that engineers spend doing maintenance work versus building new features? Such a metric would initially drop as engineers who had *planned* to take on maintenance work don't have to do so, and instead spend time building out new features. But those same engineers would face new business challenges and would make the prudent decision to take on new types of technical debt in order to provide business value faster. Just as with self-reports of technical debt, we'd expect an objective, general maintenance overhead metric to rebound as an engineering organization adapts and optimizes.

Other metrics might move more slowly, and then readjust back to a baseline for a different reason entirely. Consider self-reported code quality on a developer experience survey. We'd expect that measure to increase (more slowly than the previous metrics) as engineers clean up old code to address technical debt. Eventually, engineers might adjust to the "new normal." "Cleaned up code" becomes the new baseline by which they judge code quality. Engineers' rising expectations of what constitutes "good code" might change, and so their self-reported satisfaction with code quality might return to current (premagic wand) levels (see Figure 2).

The "regression" won't mean that we failed to improve software development in general. Rather, it is a sign of success: Engineers have embraced their new normal and have raised their expectations accordingly.

Finally, some "productivity metrics" won't move at all. It's not that these are not affected, but that the work simply shifts from technical debt cleanup to other types of work. Consider metrics like pull request throughput or lines of code produced per engineer. For these, we expect these measures would stay flat. Reducing technical debt doesn't change the amount of work an engineer does in a week; it only changes the nature of their work. Their work is transformed because they now work on *different* things, but a metric like #LOC doesn't capture that subtlety.

What About "Capital P" Productivity?

How *can* we measure the impact of our magic wand? We don't have a true measure of productivity. (Economists have worked on this for much longer than software engineers. Lots of smart people have also not come up with a productivity measure. This is simply a *wicked problem*.[7]) We would argue that if we did, we would see productivity jump up and stay there (see Figure 3).

Similar advances have occurred many times in our field: A major new technology comes along and increases productivity, but a few years later we have forgotten because it became our new normal. The productivity gains are still there though.

Here's an example from computer science history: When FORTRAN was introduced, it was called "automatic coding."[8] Take a look at this introduction. Remind you of anything? See Figure 4.

Today, it feels *absurd* to refer to an antiquated language like FORTRAN as "automatic coding." But FORTRAN was only one step of many we have made along the automation–abstraction cycle.[9] In this cycle, our field starts with a common, shared problem. As engineers explore the problem space and try out different ideas, experts emerge who identify useful solutions to

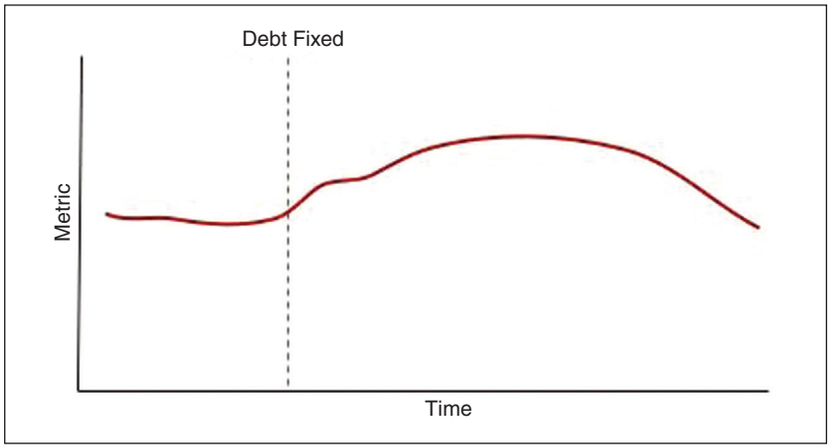


FIGURE 2. Other metrics (e.g., perceived code quality) might move more slowly, and then readjust back to a baseline for a different reason entirely.

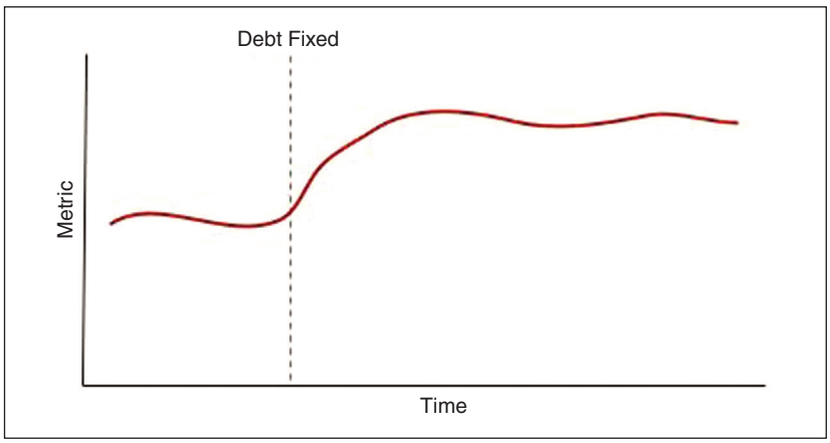
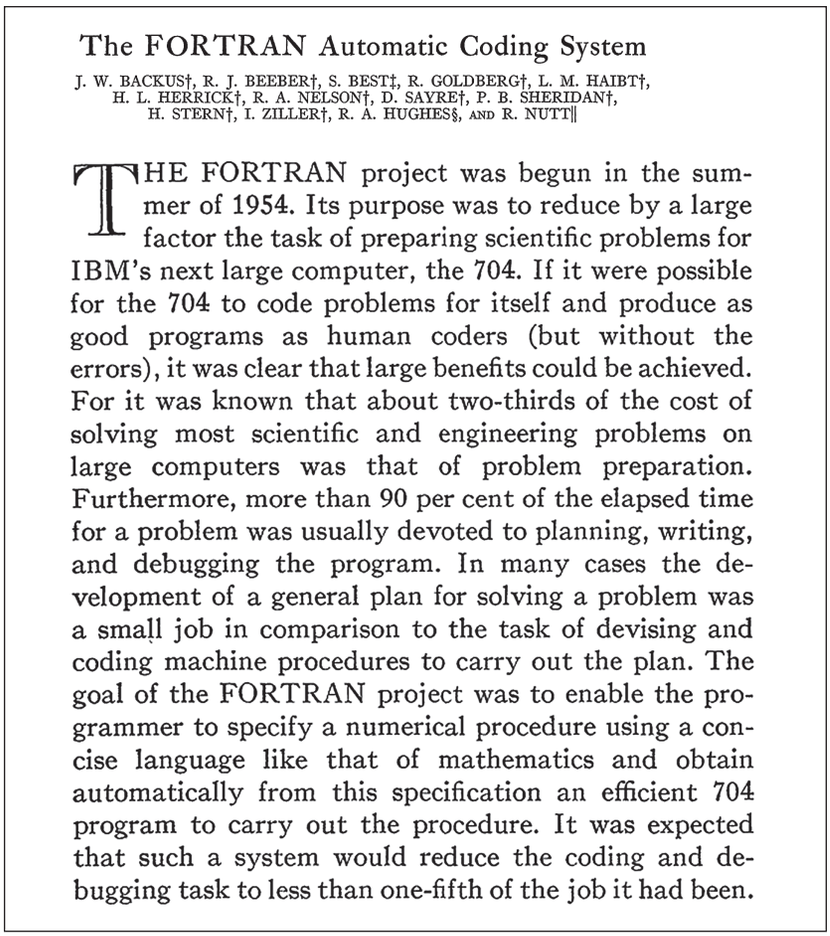


FIGURE 3. A good metric for Capital "P" productivity would be permanently boosted by the removal of technical debt.



The FORTRAN Automatic Coding System

J. W. BACKUS†, R. J. BEEBER†, S. BEST‡, R. GOLDBERG†, L. M. HAIBT†,
H. L. HERRICK†, R. A. NELSON†, D. SAYRE†, P. B. SHERIDAN†,
H. STERN†, I. ZILLER†, R. A. HUGHES§, AND R. NUTT‖

THE FORTRAN project was begun in the summer of 1954. Its purpose was to reduce by a large factor the task of preparing scientific problems for IBM's next large computer, the 704. If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. For it was known that about two-thirds of the cost of solving most scientific and engineering problems on large computers was that of problem preparation. Furthermore, more than 90 per cent of the elapsed time for a problem was usually devoted to planning, writing, and debugging the program. In many cases the development of a general plan for solving a problem was a small job in comparison to the task of devising and coding machine procedures to carry out the plan. The goal of the FORTRAN project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and obtain automatically from this specification an efficient 704 program to carry out the procedure. It was expected that such a system would reduce the coding and debugging task to less than one-fifth of the job it had been.

FIGURE 4. Automating away technical debt might be like the FORTRAN "automatic coding system"; it's just another step in the ongoing automation-abstraction cycle.

the problem. These become best practices that are shared with the wider community. Eventually, these best practices become codified into our tools, languages, and frameworks. This now raises the level of abstraction: Developers can now work faster, so they take on more complex software…and therefore run into new types of problems. The cycle starts again.

Structured programming, object-oriented programming, test-driven development, design and architecture patterns, and code review are additional examples. This has and will happen with technical debt too (see the aforementioned example about autoformatters).

If new technologies, whether AI-based or other forms of automation, are able to reduce or even *eliminate* some forms of technical debt, it will absolutely be a win for developer productivity, but it might not be visible in our productivity metrics.

If we were to speculate about a metric that *should* be responsive to our technical debt magic wand, we might pick a metric that incorporates engineering efficiency *and* business value. After all, technical debt emerges from the way that engineering capacity is allocated to balance realized business value and long term sustainability of an engineered system. Thus, it is tempting to consider a metric like revenue per engineer (RpE)[10] to measure the impact of eliminating whole classes of technical debt. If our magic wand enables us to reduce the engineering effort spent on maintenance and rework and increase the engineering effort spent on new (presumably revenue-generating) projects, then RpE should go up. Experts have already identified that advanced tooling and automation, lower-friction developer experience, and better management of technical debt are drivers of increased RpE.[10]

We feel compelled to offer an important caveat here. RpE is a reasonable metric for the efficiency of an engineering business unit but not a good measure of individual or team engineering productivity. Factors unrelated to engineering productivity may substantively change RpE. Leaders should not lose sight of these facts. An engineering organization that is productive and efficient at building things will show low RpE if business strategy, product direction, or company operations drag down realized revenue. An engineering organization that is close to collapse from friction, technical debt, and inefficiency may show high RpE if they are working in a new market space with few competitors and excellent business and product colleagues.

In sum, what happens when technical debt goes away? We think engineering organizations are likely to get more efficient. We think that this increased efficiency may show up in productivity metrics in a nuanced way that varies with the metric and probably changes over time as the organization reallocates resources and settles back to a comfortable level of technical debt-related risk. We think that metrics that reflect both engineering productivity and business outcomes are a reasonable, but perhaps fraught, choice to measure the impact of advances in how technical debt is incurred, managed, and eliminated. But this is our speculation and we are eager to observe examples out in the wild.

References

1. P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (Dagstuhl Seminar 16162)," *Dagstuhl Rep.*, vol. 6, no. 4, pp. 110–138, 2016, doi: 10.4230/DagRep.6.4.110.
2. M. Fowler. "Technical debt quadrant." martinfowler.com. Accessed on Mar. 7, 2025. [Online]. Available: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html
3. A. Ampatzoglou, A. Ampatzoglou, A. Chatzigergiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Inf. Softw. Technol.*, vol. 64, pp. 52–73, Aug. 2015, doi: 10.1016/j.infsof.2015.04.001.
4. G. Rosalsky, "Why the AI world is suddenly obsessed with a 160-year-old economics paradox," *Planet Money Newslett.*, Feb. 4, 2025. Accessed: Mar. 17, 2025. [Online]. Available: https://www.npr.org/sections/planet-money/2025/02/04/g-s1-46018/ai-deepseek-economics-jevons-paradox
5. "ISO 31073:2022(en) Risk management — Vocabulary." ISO - International Organization for Standardization. Accessed on Mar. 11, 2025. [Online]. Available: https://www.iso.org/obp/ui#iso:std:iso:31073:ed-1:v1:en:term:3.3.27
6. Y. Cai et al., "Understanding architectural complexity, maintenance burden, and developer sentiment—A large-scale study," in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, Ottawa, ON, Canada, 2025, pp. 2176–2187, doi: 10.1109/ICSE55347.2025.00168.
7. "Wicked problem." Wikipedia. Accessed on May 23, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Wicked_problem
8. J. W. Backus et al., "The FORTRAN automatic coding system," presented at the Western Joint Comput. Conf.: Techn. Rel. IRE-AIEE-ACM (Western), Feb. 26–28, 1957.
9. M. Shaw, Carnegie Mellon University about the automation-abstraction cycle in engineering, Personal communication.
10. "Revenue per engineer report," DX, Salt Lake City, UT, USA, 2025. Accessed: May 9, 2025. [Online]. Available: https://getdx.com/report/revenue-per-engineer-2025/

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.