Anticipating and Dealing with Operational Debt

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly *Site Reliability Engineering* book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura is currently enjoying a well-earned sabbatical (and tinkering with some of her own projects) after 15 years in industry, most recently at Google.
laura.nolan@gmail.com

We are all familiar with the concept of technical debt, the idea that over time, software systems become harder to change and maintain because of shortcuts taken earlier. An example of technical debt is the lack of a comprehensive suite of unit tests (or a flaky test suite). Old, unused code that hasn't been removed is another example. Technical debt can occur early in a system's lifespan as shortcuts are taken in order to launch, but most of the time the problem of technical debt gets worse as a system ages.

Operational debt is different. It happens when a system is launched, or experiences rapid growth in usage, before operational tasks are automated, leaving them to the system's human operators. In organizations with a focus on automating routine operational tasks, much of this is solved over time, leading to less operational debt as a system ages.

Technical debt is like credit card debt—acquired piecemeal over time. Operational debt is more like a mortgage: it can be paid down over time leading to ownership of a stable, well-automated system. However, sometimes people do have problems paying off their mortgages. The worst case scenario is a team that ends up with so much operational debt that they don't have cycles to work on fixing it, instead spending most of their time on *toil* [1]—tactical work that doesn't improve their systems in the long term.

This environment isn't good for engineers, and teams in this situation will struggle to retain staff.

Types of Operational Debt

There are five main categories of work that, if not automated, lead to operational debt.

One is routine housekeeping that happens on a schedule. This might include taking and validating backups, updating certificates, and making sure personally identifiable information is deleted after a certain time period.

The second is recovery from routine failures like loss of a hard drive, transient network problems, or a machine restarting.

Another category involves managing change over time. This includes things like performing migrations, doing capacity planning, and monitoring for performance regressions.

Many systems involve some routine per-customer work like setting up permissions, quotas, or other resources.

Finally, there is non-routine work that scales with your system's growth. This includes turning up new instances of your systems, dealing with abusive users, resharding datastores to deal with growth, and investigating performance issues on behalf of customers.

Operational Debt after Launch

Some amount of operational debt in a newly launched system is inevitable. This is for two major reasons.

The first is unknown unknowns—issues will crop up in production that weren't anticipated, and some of these will need automation to handle them. For example, take a system where the underlying datastore occasionally has replication issues. Sometimes a customer's changes don't get reflected everywhere, they complain, and someone has to go and unwedge it by hand. There are several potential approaches to automating this problem away, ranging from fixing the underlying replication issues to various bolt-on approaches, but either way, noticing the pattern and automating away this sort of problem takes time.

The second reason is that even for routine and anticipated operational tasks, the development of the core system itself usually has to precede development of complex automation. It's hard to automate a process for a system that doesn't exist yet.

Managing and Planning for Operational Debt

Operational debt is not inherently bad, but too much of it certainly is—again, like mortgage debt. It needs to be planned for and managed, particularly when launching a new service, instituting a major change to an existing service, or in times of fast growth.

First, track what your team is spending its time on now. If your team already has a lot of operational work, it may need to be reduced before you can afford to launch something new. At Google, SRE teams aim to spend under 50% of their time on operational work.

Next, estimate what you're getting yourself into. For your planned system or feature:

- What are the periodic "housekeeping" tasks?
- What failures or problems will the system likely encounter regularly, and how much work will it be to recover from them?
- What are the change management tasks?
- What are the routine per-customer tasks and the likely non-routine ones?
- How is the user base likely to grow over time?
- What is automated already, and how much effort is likely to be required to automate the rest?

You should also budget for some unknown unknowns. This is technology, after all.

After this exercise, you should have a better idea whether or not your team will be able to afford the launch and what needs to be automated first so that your team can remain productive.

Zero operational work shouldn't be your goal. Some tasks aren't worth automating because they're done infrequently and there won't be a positive return on the investment of time. Some operational work is novel, like debugging new problems and dealing with outages, and does require human skills. But excessive operational debt is dangerous when it soaks up so many of a team's cycles that they can't do engineering work.

Anticipate operational debt, budget for it, and keep your team out of operational overload [2].

References

[1] V. Rau, "Eliminating Toil," in *Site Reliability Engineering (O'Reilly, 2016)*: https://landing.google.com/sre/sre-book/chapters/eliminating-toil/.

[2] R. Bosetti, "Embedding an SRE to Recover from Operational Overload," in *Site Reliability Engineering (O'Reilly, 2016)*: https://landing.google.com/sre/sre-book/chapters/operational-overload/.