SRE Tools as Products

Joan C. Smith

July 2019

Abstract

As part of their work engineering reliability into services, SRE teams are often responsible for building tools. These tools might be intended for internal or external use:

- Internal tools: Scripts that help with incident response, small binaries that perform a repetitive task for toil reduction, or the infrastructure for performing software releases of a service.
- External tools: Intended for other engineers at the organization, to make infrastructure easier to use or more accessible.

I posit that the vast majority of SRE teams are building some tooling; SREs often don't think of this tooling as a product, and would benefit from doing so. Whether the team is building tools for their own use or for engineers outside the team, bringing a product mindset to SRE-written tooling can help improve both the way SRE teams operate internally and the reliability of the associated infrastructure.

When designed and built individually, the different tools that an SRE team builds can end up disjointed, hard to understand, and potentially in conflict with other tools their organization or team uses. These problems can be solved by taking an end-to-end product approach. Such an approach, for example, enables engineers to seamlessly troubleshoot an alert through multiple phases -- from a runbook, to monitoring consoles, to remediation scripts -- so all the tools work together as a single flow.

By thinking about SRE work as product work, the quality of a team's output improves, increasing the reliability of their service. In the example above, the three pieces of the SRE workflow (runbook, console, remediation scripts) become unified: SRE's work is no longer to update a runbook or tweak an alert, but to make sure the "response" flow of the product is usable, is coherent, and reduces MTTR. Conceiving of monitoring, alerting, configuration, documentation, and incident response scripts as a unified product where the primary objective is to "increase the reliability of the service" provides another strategy for SRE teams to accomplish their overall goals.

Introduction

As discussed *in Site Reliability Engineering: How Google Runs Production Systems*[1] and *The Site Reliability Workbook*[2], Site Reliability Engineering (SRE) is a specialized job function that focuses on the reliability and maintainability of large systems. As a set of engineering approaches to running

[1] SRE Book: https://landing.google.com/sre/sre-book/toc/index.html
[2] SRE Workbook: https://landing.google.com/sre/workbook/toc/

better production systems, SRE seeks to provide reliability and uptime appropriate to the needs of a service or system. SREs support product development by enabling a fast rate of change while preserving system reliability.

Within SRE, staffing cannot scale with service growth -- to do so would violate the fundamental principles of SRE's approach to engineering work, by relying on humans rather than software to manage a service's complexity. As a result, we focus on scaling SRE teams sublinearly -- doing more with fewer people. SRE teams are small and nimble allowing them to tackle traditionally overwhelming operational load through software engineering. Much of the work of these teams is aimed at automating away toil and creating engineering solutions to operations problems.

As such, much of SRE's engineering effort is funneled towards developing tools. SRE-developed tools fall into two primary categories:

- Internal tools that support the team's own work, e.g. software releases, monitoring, etc.
- External tools that define the interface between the critical infrastructure system supported by an SRE team and other engineers.

SRE-developed tools are usually built by and for systems engineers to solve the problem they're experiencing today. SREs seldom build tooling for a company's external customers, sales reps, or product managers.

When an SRE team is successful, the tools they build end up saving significant engineering time and energy across the organization. This article explores how treating SRE-developed tools as products can improve productivity, decrease toil, and reduce MTTRs not just within the SRE team, but for the whole organization.

Types of SRE Developed Tools

Internal Tools

SRE's internal tools are often the result of toil reduction projects: the team identifies that a manual process is causing repeated, unnecessary work, and prioritizes building a set of tools to make that work vanish. These tools are considered the bread and butter of SRE teams. They are "what happen[s] when you ask a software engineer to design an operations function."[3]

Examples of internal tools that SREs spend significant time engineering and building include:

- Monitoring dashboards to optimize incident response
- Software to automate mitigation of common failure modes in a system
- Microservices that perform a repetitive task for toil reduction
- The infrastructure to perform software releases for large and complex systems

[3] Quote from Ben Treynor: https://landing.google.com/sre/

These tools are used primarily (and often only) by the SRE team that builds them. These internal tools are critical to the operational sustainability of SRE teams — engineers can't manage an increasing number of increasingly complex services unless the common toilsome tasks become easier and easier over time.

External Tools

SRE teams are also often responsible for the tools that define the interfaces between other engineers and the SRE team's system. This is especially true for SRE teams that manage critical infrastructure in large organizations. For example, SRE teams often provide client monitoring dashboards for engineers building on top of a foundational infrastructure system. Using these dashboards, engineers outside the SRE team can understand performance from the service's perspective (e.g., the latency of queries to a particular user's database, or the resource usage in a downstream service caused by RPCs from the upstream service).

Sometimes these external tools are used by dozens or hundreds of engineers, and dictate how engineers interact with an infrastructure tool from beginning to end. For example, Bigtable and Spanner SRE build external tools that allow other engineers at Google to obtain database instances and grow these instances when an upstream application experiences growth.

In the ideal case, SRE teams also build sophisticated self-service debugging tools to ensure that engineers building on top of their services can debug and fix issues themselves. If users of a service have tools available to debug common problems (like insufficient capacity, throttling, or reduced availability), then they may be able to resolve incidents without needing to engage the SRE team at all. In addition, if the SRE-built tooling allows the user to make progress debugging on their own, the user will be able to provide significantly richer information about the problem, including graphs, error messages, and diagnostic output when they do need to engage with SRE. These tools protect the SRE team from an endless stream of "Your service is slow; please help!"-type questions, and ensure that when SRE is engaged they're able to act quickly based on useful information provided.

External tools can make or break an SRE team. They dictate whether your users need to ping an oncall for every change or blip on a graph, or whether they have enough information to make changes, debug, and resolve problems themselves.

SRE Engineering Approaches

Traditional SRE

Beyond tool development, Site Reliability Engineering as a discipline fundamentally relies on understanding people and human error. Without understanding how humans err, SREs can't build reliability into systems that humans manage. Google's SRE experience has proven that people who fear shaming and punishment won't come forward when they think they contributed to an outage, so everyone does our best to ensure a culture of blamelessness. When SRE culture is working as intended, an outage isn't the fault of an engineer or a user— it's the system's fault.

SRE seeks to further apply the principle of blamelessness to system design: we avoid building traps into our systems in order to make safe actions easy, and destructive actions unlikely. For example, the default behavior of a CLI tool should never wipe every edge cache in the fleet when it doesn't have any arguments. Instead, we make that CLI print out its help message by default, and only perform destructive operations when they are explicitly requested. Postmortem action items very often feature changes along these lines— the tool or system behaved unexpectedly when a person used it, and we should mitigate how often or easily that happens. By exercising empathy for a stressed out or tired user, user experience changes like these make SRE tools easier to use.

Traditionally, SRE has been quite successful at reactively identifying and solving problems discovered during outages. SRE also excels at applying systems and reliability engineering to more proactive projects — for example, running drills and disaster simulations[4] to find fixes before they bite in production. SRE might further improve upon proactive approaches to improving reliability and reducing toil by turning to product engineering.

The next generation

At their best, the internal and external tools built by SREs make other engineers in the organization significantly more productive. However, tools don't always achieve this goal. Sometimes these tools are plagued by poor user experience, missing documentation, or sharp edges that have surprising behaviors. Even the most operationally healthy teams are often plagued by little sharp edges on tools: behavior that the experienced oncaller warns newcomers about, or tricks that everyone writes on a post-it on their monitor.

These edges end up even sharper if an SRE team is experiencing operational overload. Once in this state, the tools end up decaying further— compounding the operational load, reducing engineering velocity, and overall leaving the team in a critical state. Regardless of the health of the team, the sharp edges on tools— the little gotchas recorded in a playbook, or the trick you hope every external tool user manages to remember— represent a tax on engineering effectiveness. This tax effects not just reliability engineers, but potentially the whole engineering organization that relies on that SRE team. By joining reliability engineering with product engineering, we can sand down some of those sharp edges, improve reliability, and generally reduce the tax on our engineering organization.

Defining product engineering

Product engineering has some fundamental differences from the system engineering practices with which SRE teams are intimately familiar. While systems engineers focus primarily on the interaction between software systems (often across organizational boundaries), product engineers focus on how people interact with software systems. Parts of the human/software interaction are well understood in SRE: we write blameless postmortems and do our best to engineer our systems so they are resilient to human errors. However, product engineering systematizes improving the human/software interaction through a set of defined principles.

Product engineering as a discipline prioritizes a few main concepts:

- Short feedback cycles

[4] 10 Years of Crashing Google: https://www.usenix.org/conference/lisa15/conference-program/presentation/krishnan

- Continuous iteration
- Imperfect Minimum Viable Products (MVPs)
- Ruthless prioritization based on ROI
- A militant focus on user experience
- (above all) A genuine sense of user empathy

Each of these ideas is as fundamental to product engineering as observability, monitoring, postmortems, and toil reduction are to reliability engineering.

Short feedback cycles enable engineers to understand which change to make next by creating visibility into how previous changes were received. Products aren't delivered fully-baked with every feature designed end-to-end; rather, they are built piece by piece, accounting for how users respond, and making shifts as required. These short feedback cycles go hand in hand with continuous iteration: a feature is unlikely to be done the moment it ships to production the first time. Instead, product engineers try to ship early and often. They rely on short feedback cycles to understand what needs to change in the next iteration, and continuously improve the product, stopping only when working on the next iteration of one feature provides a lower ROI than working on another big feature. Short feedback cycles, combined with continuous iteration and ruthless prioritization lead to (often) natural decisions about when something is complete. A product isn't done because it shipped; it's done because users like it. Users know what the product does, they're using the product, and it makes their experience with the overall system materially better. Each of the tenets and practices in product engineering are fundamentally connected to the idea of user empathy.

Product engineering as applied to SRE

Cultivating product thinking and product engineering expertise within SRE teams presents an immense opportunity. Many of the engineering challenges in building tools, both internal and external, amount to product engineering challenges.

For example: on one team at Google, an engineer was investigating a complex recurrent bug and wrote a script to collect every instance of the problem globally by joining information from structured logs and system metadata. The script worked brilliantly, solved the problem, and then sat indefinitely on that engineer's desktop (experimental directory). Nine months later, a similar problem began arising repeatedly. A different engineer was assigned to the task, and wasn't aware of the prior work. The second engineer repeated the first engineer's work, so a problem that could have taken minutes to investigate and resolve instead took just as long as the first instance.

One of the fundamental tenets of SRE is that the *person* isn't the problem, the *system* is. This duplication of effort and resulting increased time to repair wasn't the *"fault"* of any of the participants, but rather the fault of the system that allowed the SRE team to easily lose useful work. Moments like this represent an untapped opportunity to transform the suite of internal tools that SREs build for themselves into a set of coherent, unified, products. If instead of leaving that script to gather dust, the team had a systematic product engineering approach, it might have ended up as an additional command in a commonly used mitigation CLI for the team, complete with a help message, reasonable default behavior, and a pointer from a playbook. If the one-off script had been just modestly more "productized," the second outage could have been repaired in just a few minutes.

Of course, taking one-off scripts and turning them into tools entails tradeoffs— code that's written will always have to be maintained. Without a focus on ROI, these tradeoffs are expensive; bringing product engineering principles into SRE without bringing a strong focus on ROI is a dangerous exercise. Since Google is an extremely large organization, many SRE teams build internal tools that are vital to keeping Search, YouTube, and other critical services running, as well as external tools used by dozens or hundreds of engineers each week. For Google, often the ROI from bringing product engineering into SRE is quite high: a few minutes saved in a critical outage or hundreds of users that no longer need to file tickets represent significant returns. In other organizations with fewer users of internal tools, or systems less sensitive to downtime, the tradeoffs might mean that only the first of several iterations to make a tool easier to use are obviously worth the investment. Much of the point of a product engineering mindset is to actively make these ROI tradeoffs and ask the questions about whether a tool could or should be improved.

Building a coherent, unified internal tooling set has significant upsides. An SRE truism states that whatever is currently broken was probably broken by a recent change. Often, SREs spend much of their incident response time debugging to understand which systems changed recently, and how they changed. At Google, given the substantial layering of technical infrastructure, understanding what changed recently can be quite time consuming.

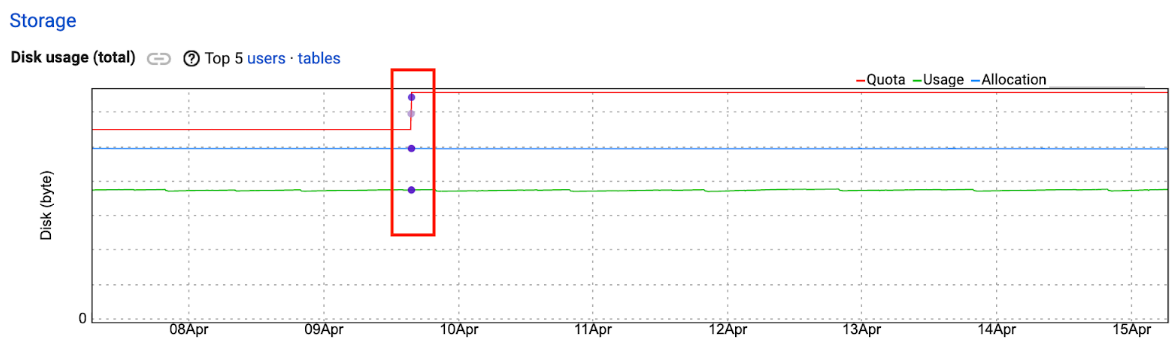
Case study: Product engineering for internal tools

Over several years, the tools that help SREs understand production changes have improved significantly. In one particular case, Bigtable SRE cut mean time to repair (MTTR) by collecting production changes, then displaying them on their debugging dashboards as additional information.

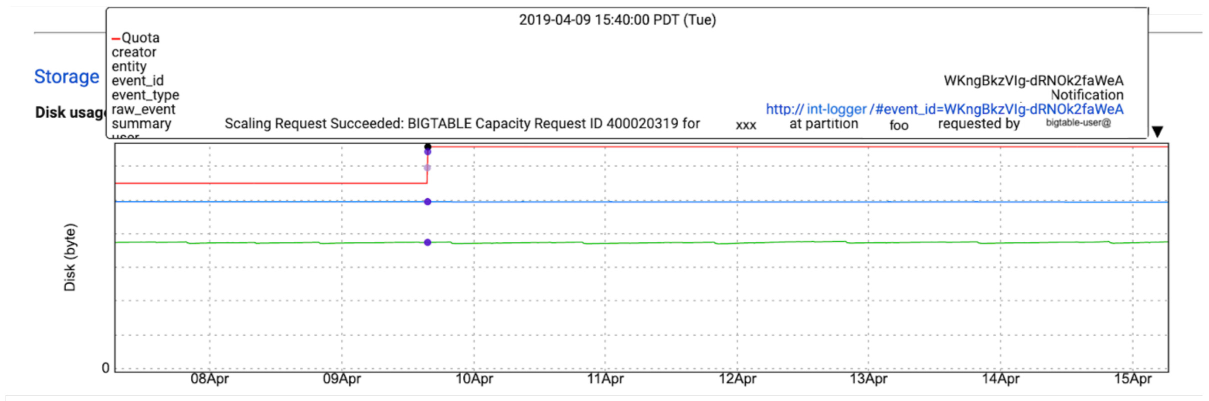
Bigtable users are responsible for their own resource allocations. In order to build something on Bigtable, an engineer works with their engineering organization to acquire the necessary compute and storage resources. The engineer provides those resources to Bigtable, which uses the resources to run a managed service. These operations are entirely self-service and usually safe, but resource changes can sometimes cause surprising behavior.

Bigtable SRE discovered that they were spending significant effort during incidents to understand why a resource change had occurred, and who had made it. In order to make this process easier, an SRE put together some quick scripts that would enable other SREs to pull the right information from a variety of systems and print it out. This tool proved to be quite useful, and significantly reduced how long it took to understand if resources played a role in an outage. This was a low cost, high ROI fix that improved the experience of being oncall for Bigtable.

After SREs used the tool heavily for several months, an oncaller decided to build upon the tool: running the tool and reading the results was great, but what if you could find the same information without manually running a tool? Upon discovering that Bigtable's regular investigation dashboards had access to all the same information provided by the tool, the engineer spent a couple of days of their oncall shift adding resource change events as additional data on the existing graphs. The resource changes were displayed as round points overlaid on the trendlines. Clicking on a point provided additional information and a link to get more detail. This graph allowed SREs to quickly identify when a resource change caused latency to significantly increase, or a database to start exhibiting overload behavior. In these cases, investigation became trivial and MTTR was significantly reduced. This technique has proven to be so useful that it has become standard practice for resource change requests and has been extended to additional point-in-time changes.



Dots indicating additional information about when disk bytes were increased.



```
2019-04-09 15:40:00 PDT (Tue)
–Quota
creator
entity
event_id                                                                          WKngBkzVIg-dRNOk2faWeA
event_type                                                                                  Notification
raw_event                                                     http:// int-logger /#event_id=WKngBkzVIg-dRNOk2faWeA
summary     Scaling Request Succeeded: BIGTABLE Capacity Request ID 400020319 for   xxx   at partition   foo   requested by   bigtable-user@
```

An overlay providing additional information about a resource change request, when a dot is selected.

This approach represents product engineering for internal tools at its best: an iterative process driven relatively organically by ROI. Someone had an idea to improve incident response, and they took the simplest approach to implement it. When that implementation proved sufficiently useful, engineers invested more time in improving the usability of the tool, making the whole flow more coherent.

Case study: Product engineering for external tools

While the opportunity for product engineering in internal tools is large, applying a product engineering approach to SRE's external tools presents an even greater opportunity. External tools that SRE builds without a product approach can be comically difficult to use. Such tools expect each user to both be a system expert *and* be able to inhabit the mental state of the tool's author at the time of creation. This disconnect is especially visible in client monitoring dashboards.

SREs building a client monitoring dashboard will often focus on just getting the graphs on the page. The graphs end up appearing in a more or less random order, even if the lone engineer building the dashboard had a mental model for why the latency graph should be above the errors graph, and why the stuck transactions graph should be five clicks away on a completely different page. When a system user needs to investigate a problem or answer a question on the client monitoring dashboard, they have to hunt through multiple pages to find a graph that looks spikey, or to find the resource usage graph buried at the bottom of what otherwise looked like a debugging page. Sometimes, these users give up and just "file a ticket" or walk over and tap the oncaller on the shoulder.

When SRE builds external tools that are hard to use, engineers on other teams aren't the only people who suffer. The SRE team ends up plagued with a large volume of easy-to-answer questions that nevertheless monopolize valuable engineering time. If SREs routinely close tickets with a link to a particular dashboard or page of documentation, they're facing a user experience failure. Why are SREs serving as sophisticated search functions or redirectors? The SRE team’s external tools need to speak for themselves: users with a question should be funneled to the documentation and dashboard they need. Applying a product approach to these tools can reduce toil and improve the productivity of engineers building against infrastructure.

At Google, infrastructure SRE teams often support a large number of engineers on other teams. A team of 14 SREs globally will support a fundamental piece of infrastructure that virtually every software engineer uses when building a product or service. These critical teams support databases like Spanner and Bigtable, fundamental resource allocation infrastructure like Borg, Colossus, and the network, and the tools for monitoring and alerting. The following example features Colossus, but this pattern occurs on every team with many users that depend on a service.

Colossus SRE identified that users regularly paged the team during emergencies to obtain permission to use additional storage bytes. These tickets were relatively quick for an SRE to resolve, but the volume of the tickets was increasing quickly as the service grew. The team prioritized a two quarter project to provide a self-service mechanism for making this configuration change, and built a CLI that could perform the relevant operation. They declared success when they launched the tool and sent an announcement email with a pointer to the tool.

After the launch, the ticket load dipped briefly, but Colossus SREs were surprised when it returned to previously high levels a month later. The tickets were easier to handle now: instead of performing a manual operation, an SRE just had to respond with a link and some instructions. This was still not a great use of time, especially in emergency situations. Fixing the continuing ticket load required a bit of product engineering. Specifically, SREs needed to ask:

- Who are the users that want the change?
- When did they discover they wanted the change?
- How did users know they needed the change?

It turned out that users discovered they were running low on resources either from an alert or from proactively looking at a dashboard — when they saw one line getting close to the other on the dashboard, they paged Colossus SRE. Colossus SRE identified this pattern and made a couple improvements: they added instructions to the dashboard, in addition to a button that ran the tool and immediately granted a small quantity of emergency resources. This button completely removed Colossus SRE from the emergency requests, significantly reduced MTTR, and improved the Colossus user experience. The CLI tool on its own didn't remove the toil, but adding a button meant that Colossus SRE no longer had to respond to emergency resource requests.

Moving an SRE organization towards product engineering

As shown in both of these case studies, Google SREs improved several of our services organically—having a few SREs with product development backgrounds allowed cross-pollination between the engineering disciplines. It isn't yet standard practice to ask product engineering questions on SRE design docs, or prioritize user experience iteration on tools development projects, but teams that have started adopting these practices have been able to scale better, improve MTTRs, and reduce toil. Recently, externally-facing products like Stackdriver IRM (Incident Relationship Management) have entered into unchartered territory for Google SRE by taking product engineering into account from the earliest design phases. More of our internally-facing tools may move towards a more formalized product engineering approach as it gains success.

Bringing product engineering into SRE does require some cultural change. There's nothing so permanent as a hack in production, and a series of these expedient hacks, built up over time, contributes to the sharp edges and unfriendly tools described above. SRE has succeeded in part by hunting down and implementing least effort in the short term fixes that reduce toil and increase reliability. The shift to product engineering approaches requires a change in mindset. Product engineering takes a different approach: it prioritizes reducing the global effort over time by using a more expansive view of who's effort counts, and what effort adds up to unsustainable toil. Product engineering should be part of the strategy that allows SRE to continue delivering outsize value.

While product engineering does represent a cultural shift, it's one that can begin in small ways, from the bottom up. SREs can begin developing a sense of the user experience they provide with their tooling, and begin to adopt product engineering techniques into existing every day workflows— for example, in design doc reviews, when creating postmortem action items, and when improving dashboards. By considering how users interact with the systems and prioritizing short feedback cycles, continuous iteration, and positive ROIs, SREs can improve the usability, stability, and performance of critical infrastructure they support. These changes might begin either with an individual SRE, or with an SRE-wide movement. Just getting in the habit of asking product engineering questions and then iterating on answers brings many of the benefits explored in this article.

If you're an SRE considering implementing some product engineering approaches in your own organization, you might try starting small. Look for a heavily used tool with some suspiciously sharp edges, and pick a relatively low investment way to sand them down, either for your own team or the people who depend on your system. Sometimes adding a warning message for the unusual case or a link to documentation is enough to make something a bit better. Channel your users (maybe even spend some time looking over their shoulders!) and see which of your tools and dashboards give them the most trouble. No need to adopt a new Javascript framework, just find a way or two to make your tools easier to use for their users. When looking for places to apply product engineering, ask yourself who your users are, what state they're in when they come to your tool, and how you can make their task a little easier. In our experience, just a few well-communicated examples of product engineering reducing MTTR are enough to increase the number of engineers applying product engineering techniques to their day-to-day work.

Lessons Learned

- The discipline of Site Reliability Engineering prioritizes:
    - Building a culture of blamelessness
    - Removing operational load with small, nimble teams of software engineers
    - Reducing failure modes by relying on software instead of humans
    - Embracing and managing risk

- SREs tend to build two types of tools: those for their own team (internal) and those for other engineers within their organization (external).
- When the user base of tools and the way users interact with tools are fragmented, the organization pays an often unseen complexity tax.
- The discipline of product engineering prioritizes a few main principles, including:
    - Short feedback cycles
    - Continuous iteration
    - Minimum Viable Products
    - Prioritization based on ROI
    - Focus on user experience
    - User empathy
- Applying an end-to-end product engineering approach to traditional SRE tasks allows organizations to improve productivity, decrease toil, and reduce MTTRs.

Related Work

What Is SRE?
Site Reliability Engineering: How Google Runs Production Systems
The Site Reliability Engineering Workbook
The Principles of Product Development Flow: Second Generation Lean Product Development
The Goal: A Process of Ongoing Improvement
Lean from the Trenches: Managing Large-Scale Projects with Kanban

Summary

This article explores the benefits that arise from treating SRE tools as products. We investigate the varieties of tools that SRE teams frequently develop, and explore how fragmentation and underinvestment in those tools can lead to sharp edges that incur a productivity tax in an organization. Then, we tackle the discipline of product engineering, defining it, examining some of its key principles, and exploring how those principles can be applied to SRE tools to shave off some of those sharp edges.

We present case studies from Bigtable and Colossus SRE to illustrate the benefits that accrue from straightforward application of a product engineering mindset by teams that support many users. The article concludes with a final exploration of ways to integrate product engineering into SRE tools, beginning with small, bottom-up actions.

In summary, several teams at Google have found significant benefit from making a mindset shift to treating their own tools as products. This experience suggests that integrating some principles from the discipline of product engineering can allow SRE teams to further reduce toil, improve reliability, and reduce MTTR for the whole organization.

Biography

Joan C. Smith is a Staff Software Engineer in SRE at Google New York. She leads a team of engineers, building reliability into the critical storage services at Google, including Spanner, Bigtable, and Colossus. Previously, she worked at Apple, Twitter and Oracle. She graduated from the Massachusetts Institute of Technology with a B. Sc. In Physics.