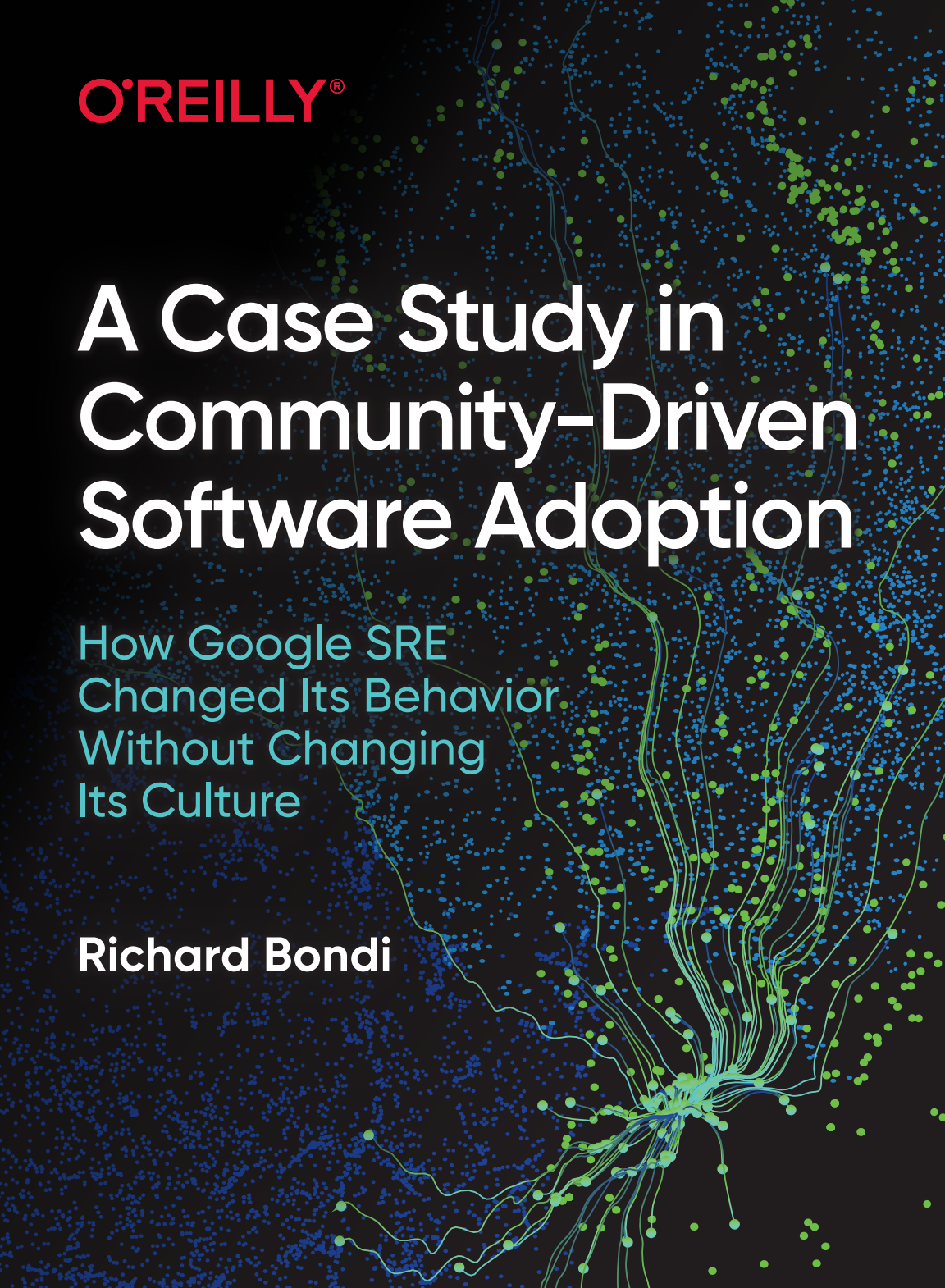

O'REILLY®

A Case Study in Community-Driven Software Adoption

How Google SRE Changed Its Behavior Without Changing Its Culture

Richard Bondi

REPORT

A Case Study in Community-Driven Software Adoption

How Google SRE Changed Its Behavior Without Changing Its Culture

Richard Bondi

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

A Case Study in Community-Driven Software Adoption
by Richard Bondi

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Development Editor: Virginia Wilson
Production Editor: Deborah Baker
Copyeditor: Octal Publishing, LLC
Proofreader: Christina Edwards
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2019: First Edition

Revision History for the First Edition
2019-06-19: First Release

978-1-098-11456-5

[LSI]

Table of Contents

A Case Study in Community-Driven Software Adoption

> *And I saw Sisyphus at his endless task raising his prodigious stone with both his hands...and the sweat ran off him and the steam rose after him.*
>
> —Homer, *The Odyssey*, Book XI

Google has had a difficult time getting Site Reliability Engineering (SRE) teams to standardize on tools and processes. We're not unique in this regard. When an SRE organization reaches a certain size, two things happen: standardization begins to matter, and you discover it's difficult.

Over a 10-year period, Google's SRE organization tried to get all of its teams onto one standard rollout tool, chosen from the dozens of internal tools that were available. The organization tried different approaches, designating a different tool as the "standard" year after year, with varying degrees of success.

Meanwhile, Sisyphus—a community-driven rollout tool that the organization didn't officially support—quietly became the de facto standard. Not only did Sisyphus have no leadership support, it was actively discouraged. Sisyphus's creators themselves said its code quality was poor and its development had been bewilderingly chaotic. It had no schedule, no funded staff, no project management, and the design documentation was written months after the tool was operational. Yet after nine years of the dozens of different automated rollout tools developed at Google, Sisyphus was used by virtually all SRE teams and many developer teams.

As part of an internal history of automation at Google, we think we discovered why. We found that any tool during this time faced two

enormous challenges that could prevent it from being adopted by many teams. One challenge was environmental: a severe shortage of time, resources, and people that SRE organizations will always face at some time. The other challenge was cultural: a resistance to enforced standards that was especially prevalent among SREs. Only Sisyphus was peculiarly adapted to overcome these challenges.

Over this nine-year period, the SRE organization was trying to change both its own behavior and its culture. In terms of behavior, almost every team wrote its own custom rollout automation. SRE culture was one of autonomy: every team was able to decide its own destiny, which meant determining the best tools for a given job. Only Sisyphus was designed in a way that allowed teams to adapt it to the particulars of each team, such as workflows, processes, and other custom tools. Teams could adapt to Sisyphus in order to adopt it, instead of the other way around. Sisyphus allowed teams to change their behavior without changing their culture of autonomy.

This case study shows how Sisyphus was able to proliferate across SRE teams. It examines the data demonstrating Sisyphus's adoption success, the two challenges Sisyphus faced, and, finally, how Sisyphus overcame both challenges. By sharing this story, we hope to offer you an example of how a tool influenced SRE behavior by adapting to SRE culture—an approach that very well might work for other organizations attempting to effect some type of change.

The Mystery of Sisyphus

In the winter of 2017, a longtime Google SRE, who we'll call Pat, stumbled onto a mystery. He had recently become the tech lead (TL) of an internal deployment program developed nine years earlier, named Sisyphus. Sisyphus is a tool designed to automate multistep deployments to Borg. Google's Borg system is a cluster manager. It runs hundreds of thousands of processes (called jobs) from many thousands of different applications across a number of clusters, each with up to tens of thousands of machines.

Sisyphus is basically a scheduler that runs other programs. Common steps include configuring and/or building a binary, draining and redirecting traffic with the Google Software Load Balancer (GSLB), testing, canarying, pushing, and rolling back if something goes

wrong at any point. Done manually, each step requires many CLI[1] or other commands. Sisyphus can execute each of the steps for you, reducing the entire release process to about a dozen clicks.

Figure 1-1 illustrates the basics of the Sisyphus interface. Here, a binary candidate was built for QA, and then Sisyphus stopped to ask whether it should proceed to canary the binary. If the human interacting with Sisyphus agrees, Sisyphus will carry out canary steps that would be toil for a human.

Figure 1-1. Screenshot of the Sisyphus interface

Although many release automation tools were written around the same time as Sisyphus, very few of these tools spread across multiple teams. Anecdotally, Sisyphus became the one rollout tool adopted by virtually all SRE teams and many software developer teams within Google.

By examining adoption data, Pat confirmed the anecdotal adoption success story. As shown in Figures 1-2 through 1-5 in the following section, year after year, many teams began to use Sisyphus and continued to do so. Yet this adoption data defied all reason: Sisyphus violated most generally acknowledged best practices of software development. Its code quality was less than stellar. Moreover, it

1 A CLI is a command-line interface tool. Instead of a user interface, you use the tool from a command line.

never had the support of SRE leadership. And yet, practically everyone was using it.

The mystery was: Why?

The Data Demonstrating Adoption

Luckily for Pat, a few factors make it easy to track Sisyphus usage via commits to individual teams' Sisyphus servers (as opposed to core and shared Sisyphus code):

- Using Sisyphus requires an SRE team to build their own version and check it into the source control system.
- Because it takes a fair amount of effort for an SRE team to build their own version, it is unlikely you would ever check in Sisyphus code unless you were really using it.
- With the exception of initialization flags, SRE teams don't customize Sisyphus by configuring it. Instead, they write additional code—typically by extending its core classes—and submit the changes to version control. Because deployment steps and commands for a particular piece of software change frequently, each piece of software likely has at least a few submits to Sisyphus a year.
- All Sisyphus code must be checked in to a specific location in Google's codebase. Therefore, it's easy to identify and count commits to Sisyphus. It's also easy to differentiate between commits to individual teams' Sisyphus servers versus commits to core and shared Sisyphus code.

Because of these quirks in Sisyphus's design, the overall number and rate of commits to Sisyphus (Figure 1-2), qualified by types of commits (Figure 1-3), is a reasonable indicator of its adoption.

A growing rate of commits year over year might indicate that a), teams were adopting Sisyphus for the first time, and b), teams were continuing to use Sisyphus after they initially adopted it.

And, indeed, data analysis revealed a trend of *growing commits*.

Code changes are committed to Google source control as a *changelist* (CL). Figure 1-2 shows that even though the *growth rate* of commits was slowly declining, for most of Sisyphus's life, the cumulative CL count was closer to doubling (100%) than not.

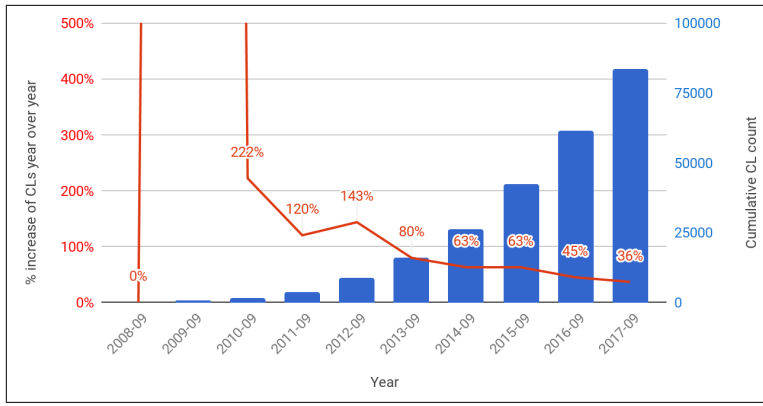

Figure 1-2. Sisyphus cumulative annual CL count over nine years

Figure 1-3 shows this growth in commits came almost entirely from individual teams' Sisyphus servers rather than core and shared Sisyphus code.

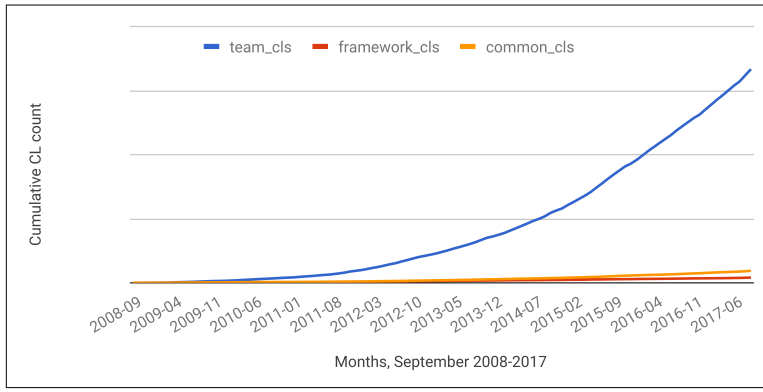

Figure 1-3. Sisyphus cumulative CL count by CL type

Figure 1-4 shows that the ratio of CLs-to-authors stayed roughly constant even as the number of CLs grew, indicating that this growth didn't come from a small band of prolific authors. (This strongly suggests Sisyphus was also dispersing across teams, but there wasn't any team-specific data to check.)

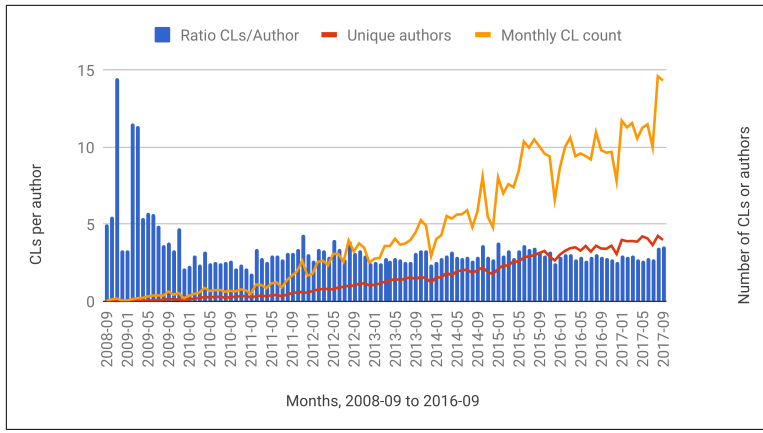

Figure 1-4. Sisyphus CLs per author

Figure 1-5 shows that the number of unique users making Sisyphus commits grew roughly linearly until 2013Q2, after which it eventually leveled off.

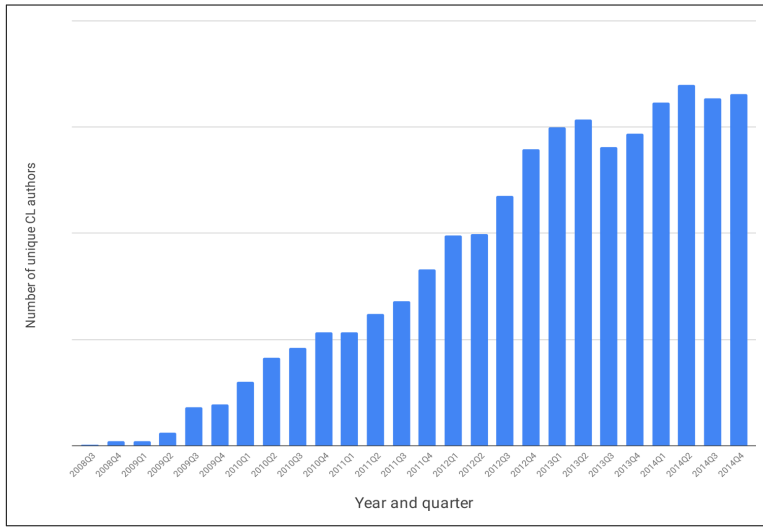

Figure 1-5. Number of unique Sisyphus committers by quarter

There are many possible distortions to check for in the data. For example, in any year, a single team might have created an order of magnitude more Sisyphus servers, whereas no others did. Or a single team could make many automatic commits to a generic Sisyphus

directory. Pat didn't have time to look for all such distortions, but these and similar distortions were negligible.

Taken together, this data strongly suggest that Sisyphus constantly spread to more and more teams, which all continued to use Sisyphus after adopting it.

The Mystery

Much of what we know about Sisyphus tells us that it shouldn't have been adopted so widely. Sisyphus's widespread adoption is mysterious for at least three reasons.

Best practices

According to conventional wisdom about software development, Sisyphus should have been an abject failure. From the very beginning, and for most of its life thereafter, Sisyphus had no roadmap, no schedule, no charter, no product requirements document, no service-level objective (SLO), and no defined development process. Its design document was written months after Sisyphus became operational. Sisyphus also had no product manager or project manager. As we will see later, SREs just worked feverishly on Sisyphus when they could, stealing time from their "real" work. Whatever explains Sisyphus's adoption success, it cannot be best practices of software development: Sisyphus violated them all.

Code quality

Even the original Sisyphus engineers acknowledge that its code is...well, not exactly a paragon of good quality. Sisyphus has the serious problems of every large software project written in Python, especially 10 years ago. The main problem with Python in 2008 was the lack of type safety. In a typesafe language, an integrated development environment (IDE; or, with more difficulty, `egrep`) can easily identify all potential callers and callees of a method and all call paths —even if the code doesn't compile. Additionally, the compiler can catch many types of bugs. In contrast, during the time Sisyphus was spreading, Python had no type checking or related compilation; and even today, programmers must guess at potential call paths. As a result, many bugs are discovered only at runtime. It can take 10 CLs to fix a bug in Sisyphus because in many cases, users can detect only

side effects and manifestations of the bug—and of its fix!—by actually spinning up a Sisyphus server and using it.

This problem is tolerable in smaller projects, but as Figure 1-6 shows, Sisyphus didn't stay small for long. As Python code increases line by line, it becomes increasingly unmaintainable and less elegant because it becomes ever more difficult to refactor. Of course, more lines of code means more bugs. And Sisyphus had a lot of bugs.

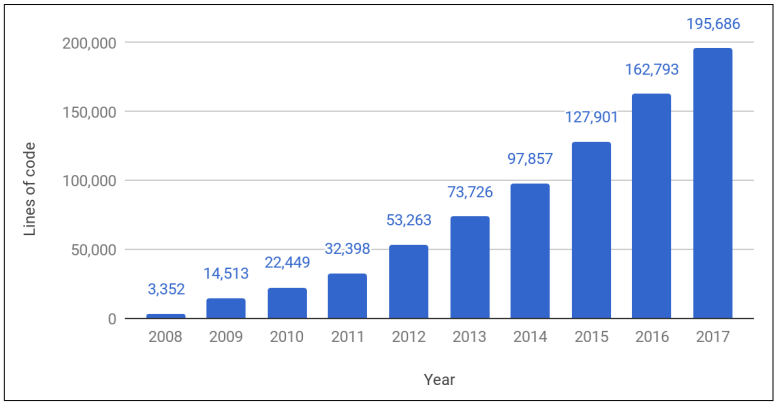

Figure 1-6. Number of lines of Sisyphus code by year

So we can't explain Sisyphus's adoption success as the result of either the quality of its project management and processes or the quality of its codebase. That these two reasons were *not* part of Sisyphus's success is puzzling. But a third reason makes the mystery confounding.

Defying death

Sisyphus was originally created in late 2008 by Search SRE, the team responsible for the heart of Google's operations: continuously deploying updates to Google's famous search engine. As Sisyphus caught on, Search SRE managers felt it was unfair that their SREs were effectively doing Sisyphus development work for many other SRE teams. Management spun up a replacement project. Search SREs were told to stop working on Sisyphus for one year because, by the end of it, the new project would replace Sisyphus for everyone. The year came and went with no replacement project.

So the Search SREs got back to work on Sisyphus, but they kept being warned of new replacements. One SRE wrote on a whiteboard behind his desk: "Sisyphus will be replaced by" and the name of the

latest replacement. When the replacement didn't materialize, he crossed it out and wrote the next replacement under it. Over the next few years, the list grew. (Sadly, no picture of the whiteboard has survived.) In addition to "official" replacements, many other SRE teams developed custom automation tools for themselves. None of these tools achieved the same adoption as Sisyphus, and some were displaced by it.

For almost 10 years, a combination of forces tried to do away with Sisyphus, but Sisyphus not only survived, it thrived.

Why?

The Answer

Explaining why Sisyphus became so popular requires digging into its history, which the following three sections explore.

The first two sections describe the two most important challenges to Sisyphus and competing tools; the third describes how and why Sisyphus was able to overcome the challenges.

Challenge 1: The Red Queen

> *"A slow sort of country!" said the [Red] Queen, "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"*
>
> —Lewis Carroll, *Through the Looking-Glass, and What Alice Found There*

Alice's Adventures in Wonderland and its sequel, *Through the Looking-Glass*, are nineteenth-century children's books famous mainly in English-speaking countries for their imaginative paradoxes of language and logic. For example, a chess piece named the Red Queen has to run as fast as she can just to stay in one place. The Red Queen has become a metaphor in science and literature, and, as such, it also captures what being a Google SRE was like in the years 2007 and 2008.

The Pyramid Scheme

In 2007, there was a cynical joke that SRE was a pyramid scheme. SREs applied to Google, eager to work on interesting and challenging problems. Instead, they went through a year of continuous,

exhausting, and repetitive Ops work. Add on-call pages, tickets, and bugs, and like the Red Queen, new SREs felt that they had to run as fast as they could just to stay in place. It could take six months or longer fielding the continuous toil of releases and tickets until you were allowed to go on-call. Then, finally, you had "free" time to work on engineering projects like creating automation—unless you were paged. On-call was the apex of the pyramid, and a new SRE was slipped into the base to replace you. But the overall reality was that all SREs were operating at a fever pitch. On many teams, a state of perpetual operational overload meant that finding meaningful work to do, or the time to do it, had become difficult.

At this time, there were about 300 SREs in Google's Mountain View, California, headquarters, split into teams of roughly 10. They faced some enormous problems. Most services were still growing at a remarkable rate, and a shortage of both people and machines resulted in operational overload. Even though SREs theoretically *could* address the excessive toil and manual labor of their Ops work by writing tools to automate toil, the volume of Ops work made finding the time to do so intensely difficult.

The Endless Cycle of Toil: Search SRE and Gryphon Rollouts

For a representative example of the endless cycle of toil, consider the circumstances of Search SRE and the main service they supported, which we'll call Gryphon.[2] Deployment of a new version could take weeks of back-to-back work because it required rolling out the new version to dozens of clusters in geographic zones across the globe. Each cluster rollout was a massive undertaking, and could take a day or more. Rollouts involved grueling toil—toil that was demanding and required skill to handle tasks like interpreting dashboards correctly, knowing how to roll back from any point, and so forth. As soon as a deployment was complete, the SREs had to immediately start all over again: the next binary version was already waiting.

This brief description does not capture the breathtaking amount of work and stress that these rollouts involved. To convey the difficulty and frenetic pace of SREs' Ops work, we must dive into technical

2 A fictionalized codename.

details. What follows is a closer look at just three of the many tasks a Search SRE rollout required: continuous configuration changes, drains, and load testing.

Each task sounds simple. But the Red Queen is in the details.

Drains and redirecting traffic

A relatively simple rollout at Google follows a common pattern. While the old binary is receiving all traffic, you push the new binary to the same cluster, but without sending any traffic to the new binary. After various testing and canarying steps, you gradually drain traffic from the old binary and then redirect traffic to the new binary.

There is no rush to drain and redirect traffic, because this rollout is just for your app; typically, your cluster hosts many unrelated apps. The number of your users is small enough that you can temporarily redirect them to your one or more other clusters or even smaller work units.

New Gryphon binaries, however, could not run at the same time as the old binaries, and each Gryhon binary required an entire cluster to itself. If a version of Gryphon was running in a cluster, no other Google apps could run there, nor could any other version of Gryphon. This was a limitation of the complex search algorithms at scale that the Google search programmers weren't able to overcome at the time. Instead, SREs had to do the following:

1. Completely drain all traffic from the target cluster.
2. Replace the old binary with the new binary.
3. Conduct testing.
4. Slowly redirect traffic.

This all had to be done as quickly as possible because your entire cluster was down most of the time, and Google had severe capacity shortages. Each cluster took at least a day—longer if something went wrong.

Continuous configuration changes

After a new Gryphon binary was built, it remained unchanged during the weeks of its rollout. But its configuration changed again, and again, and again.

Search SRE never knew which cluster it was going to roll out to until shortly beforehand. When you started the rollout on day one, you might have planned to use cluster xx on day three. On day three, after a lot of work, you might decide instead to use cluster yy, deviating from your original planned sequence.

First, you used Perforce[3] to synchronize all your planned cluster configurations with the current ones. After doing some back-of-the-envelope capacity forecasting by manually comparing graphs from today and from a week or so before, you recalculated capacity for any changed clusters and then picked the best cluster available that day. Based on your forecast of Google-wide traffic, you also chose the best time to start the push. Certain types of rollouts were even more complicated because the pushes formed a dependency tree: you had to take the order of the pushes into account.

Because of this continuous, ad hoc capacity planning, Search SRE only checked in production configuration changes after the change was in production. That way, a Perforce synchronization could instantly tell you the state of the world for the clusters you were about to affect.

Most other SRE teams did the reverse. They checked in their intended configuration changes and then pushed them to production. These teams did their capacity planning and configurations, submitted them to Perforce, and then rolled out everything. Their individual pushes didn't last long enough for production or traffic load to change significantly. For them, a rollout took maybe a few hours—not the days of Search SRE's rollouts.

Load testing

For Search SRE, load testing was not a simple matter of, "OK, let's pound it at 100% of expected max burst queries per second [QPS], and if it doesn't fall over, proceed." Instead, there were many intricate load tests.

There was the "2x load test" to detect cascading failures. A cascading failure had actually occurred once for Search. A cluster fell over under surging 2x load, and when the load was automatically

3 A commercial version control system, and one of the main version control systems Google used at the time.

switched to another cluster, *that* cluster fell over under the 3x load. The second cluster's traffic was automatically diverted to a third cluster, creating a 4x load there. Some very fast and fancy SRE footwork prevented a cascade ending in a global Search outage. As a result, every Search push now had a 2x load test. If a healthy cluster could withstand 2x, a cascade was unlikely to start. There were progressive load tests, too: latency and the rate at which load increased also mattered.

There were also complex load tests of the search caches. By design, when load surged, the search results degraded to cope. That way, all users still saw search results; the results just weren't as recent. The cache was supposed to automatically refresh when load decreased again. How cache states changed also depended on which subsystem failed or was overwhelmed by a surge, and that required additional tests.

The rollout to every single cluster entailed all of these load tests and more. Search SRE had written its own custom load test tool, but a human still had to run the tool. Load testing was very stressful because humans staring at graphs—possibly for hours, while getting the cluster up as quickly as possible—were responsible for detecting failures.

A Ray of Hope: ReleaseItNow

Then, in late 2007, a ray of hope for simplifying rollouts appeared. Three non-SRE engineers began work on a new product we'll call ReleaseItNow.[4] Their goal was for ReleaseItNow to become the tool of choice for the most common types of rollouts, which accounted for about 80% of Google's rollouts. ReleaseItNow users would include SREs plus many developer teams and other engineering teams.

By early 2008, ReleaseItNow development was making good progress, and an SRE from Search, who we'll call Cory, decided to join its development team for what seemed likely to be the final quarter of development. He wanted to make sure the finished product could take on Search SRE's most labor-intensive rollouts.

4 A fictionalized product name.

Cory's impression was mixed. On one hand, he felt that in many ways ReleaseItNow was extraordinarily well designed. It would also be the first tool to automate pushing to multiple clusters.

ReleaseItNow made assumptions that were reasonable for many releases at Google, but less so for the complex releases that required SRE teams. It assumed the following:

- Rolling out a binary to a single cluster would take about an hour.
- Testing was simple and quick.
- The order in which you rolled out and started up a binary's components to a single cluster didn't matter.
- The order in which you set up clusters didn't matter.
- A user never needed access to Google's code repository.
- Load testing wasn't critical.
- Automating drains wasn't critical, because drains for ReleaseItNow-managed rollouts would be simple affairs.

None of these assumptions was true for Search SRE.

ReleaseItNow's design made it very difficult to add if-then-else actions based on real-time conditions, drains, load testing, and other features. ReleaseItNow was a state machine, but the states were fixed: the design didn't expect that its users would ever need to add new states. And the list of additional states Search SRE needed was very long. Cory decided to focus only on supporting rollouts for one of Search SRE's use cases in ReleaseItNow's first version. Working with the team, he was able to add some callbacks inside ReleaseItNow's states for some of Search SRE's needs. That engineering task was difficult, the hooks couldn't be generalized, and the effort took six months.

When ReleaseItNow shipped in the fall of 2008 and Cory returned to Search SRE, he and the other Search SREs concluded that modifying ReleaseItNow any further was too difficult. Except for some minor help from ReleaseItNow, they were back where they started.

Red Queen on Fire

In December 2008, soon after Cory returned to Search SRE, the Red Queen suddenly began running twice as fast as before, slipping and stumbling alarmingly, as if her hair was on fire.

In addition to the large search service they currently supported, which represented most of Search SRE's operational load, Search SRE was handed an additional search service. The number of clusters to push doubled, but Search SRE staff didn't.

Google's data centers didn't double, either. Google began to suffer unpredictable and frequent capacity crunches. Capacity would run so low that sometimes engineers had to perform pushes on weekends to avoid user-facing effects. (This pattern wasn't limited to services run by Search SRE.)

The implications for a tool like Sisyphus, which was in its nascent form at this time, were very serious. Now any release automation tool built for the majority of SRE had a new and overarching challenge: a racing, staggering Red Queen on fire had to be able to adopt and use it on the fly. For example, this meant that any tools with a steep learning curve were out of the question.

In short, any SRE-wide release tool would need to overcome the Red Queen on fire.

Challenge 2: The Curse of Autonomy

It is a paradox that although SREs share a common culture, different teams do similar work in radically different ways—and are prickly about it. To appreciate how Sisyphus overcame this barrier, we first need to understand the barrier in detail and why it exists.

Team Differences

SREs are well aware of these dramatic differences between teams. One SRE described his move to a new team as entering a *Star Trek* Mirror Universe. The new team was like a twin of the old team, but in a parallel universe where everything and everyone looked the same but were fundamentally different.

Search SRE and Traffic SRE are particularly good examples of this dynamic because for years they were the same team. They began a gradual split in 2006 only because their software stack became too

complex for one team. By 2008, it had become a law in Search SRE that you did not check in a production configuration until that configuration absolutely reflected what was on production. In other words: push, then submit. This was holy writ. Traffic SRE, on the other hand, first checked in their intended configuration, then pushed it—a mortal sin and heresy in the Search SRE world.

After Traffic and Search split into two SRE teams, they also evolved to have extraordinarily different views of how a service should be run. In Search SRE, there was a relentless pursuit to make all running instances have the same amount of RAM, CPU utilization, disk space, and so on, no matter what cluster they were running in. This profile of resource usage was called a service's *shape*. For each type of hardware (e.g., Intel versus AMD, different chips speeds) Search SRE worked out the tightest possible shape. Of course, the shape changed with every new version of Search, every few weeks. Optimizing shape was extra work, but well worth the effort. Not only did this optimization save RAM and other resources, it also simplified debugging in an emergency. If your service fell over, you knew right away it wasn't because you chose the wrong shape—the shape had been carefully optimized already. Your debugging could ignore shape and focus on other possible causes.

Traffic didn't handle services that way at all. For it, the shape and architecture were something to maintain, and to reevaluate periodically. If the binary ran well with a given shape, Traffic kept it that way. The team would periodically sit down and look at graphs to see whether the shape needed to change for performance or other reasons, but otherwise it didn't spend much time optimizing for shape.

On these and other matters, SREs from the two teams tacitly agreed to disagree. As one Search SRE put it, "If you thought the other team was wrong, you didn't try to convince them. You didn't want to criticize them and make them mad. What you wanted to do was let them do it their way, but continue to do it your way."

Team Autonomy

NOTE

At a huge company like Google, generalizations about teams are unlikely to be universally true. The patterns I describe next are based on my experiences with teams over the years and the anecdotal observations of other engineers. They are not the result of any systematic survey.

These prickly differences between SRE teams results from the extraordinary autonomy we give them at Google, compared to the engineers on very large projects like Gmail. The latter engineers are sometimes abbreviated as DEVs, to distinguish production developers from the software developers who are site reliability engineers, or SREs. A 2017 postmortem of a joint SRE/DEV project identified this difference as one of the main reasons the project failed:

> There is very little notion, in this SRE team especially, of an IC [Individual Contributor] being given a task without also being given the decision making for that task. Consequently, ICs are highly reluctant to override each other, just as managers are highly reluctant to override their ICs.
>
> In the software engineering team, decision making is structured more near the top of the organization. Key decisions, even highly technical ones, are made at a leadership level, and propagated down through designated key influencers. If you are not a key influencer, you are expected to vet choices through a key influencer.

In this joint project, an SRE would think they'd agreed on a design decision with a DEV IC and then be shocked to discover later that a DEV manager had overridden it. Conversely, a DEV IC would think they'd agreed on a decision with an SRE manager and then be aghast to discover later that an SRE IC had overridden it. Each team thought the other team didn't have its act together. In fact, DEV and SRE just have different cultures.

I personally experienced this culture shock in 2016 when, as a DEV, I joined an SRE team for six months.[5] This SRE team deployed and monitored many, many different abuse-fighting systems for all of Google, designed by different DEV teams and running on different

5 Google has a program for DEVs called Mission Control, which allows a DEV to join an SRE team full-time for six months.

platforms. The SRE team atmosphere was starkly different from some other DEV teams. It was as if some years ago Larry and Sergey had handed their keys to a few SREs and said, "Here you go: these are all the abuse-fighting systems for all of Google. Please figure them out, and keep them running. The micro kitchen is down the hall on the left. We trust you completely; that's why we hired you. Thank you, and welcome to Google!"

As a team, these SREs were completely in charge of their domain. They had full autonomy; they decided on engineering solutions, internal training materials, emergency protocols, quarterly goals, new tools—everything. Managers were engineers, doing the same rotating on-call with pagers as the SREs, but with longer planning horizons, and they managed people in addition to doing technical engineering work.

DEV teams for very large products didn't have that level of autonomy. If Gmail's features were decided and implemented by 10 small teams, each with its own ideas about what language, tools, and features to implement, Gmail would be a crazy patchwork that would quickly fall apart (if it ever came together). That is why Gmail and other DEV teams for very large products need a culture of hierarchical management, with overall design and feature decisions made higher up.

In addition to designing reliable systems, SRE teams are also emergency doctors. You can think of an outage at Google as a chunk of production lying on a gurney, in convulsions and losing blood, rushed to an SRE team's desks. The SRE team has to figure out what is wrong, how to stop the patient from dying as they figure out the problem, and then how to treat the patient—all as quickly as possible. Every outage is a different, urgent, ad hoc mystery. There's no "design" that works for all of them. So, if you start ordering an SRE around—telling the team what tools they should use, or how they should use them, or what to do—the SRE can become understandably impatient with what is perceived as backseat driving.

I once asked the SRE TL of one of Google's core systems what he would do if someone very high up in SRE (who I named) were to order him and his team to switch to a new automation tool (which I also named). The TL looked away thoughtfully for a moment, and then looked straight at me and said, "I'd ignore him."

That's SRE in a nutshell. Google in general, and SRE in particular, function more like open source companies than hierarchical corporations. You can't order Googlers around; you can only persuade them.

But that's also why SREs get so much done. Paradoxically, one consequence of this autonomy is that it's extraordinarily motivating. For example, SRE teams are extremely driven, even though nobody "higher up" tells them what to do or how to do it. This gift of autonomy makes people fiercely independent. It also makes people overly sensitive about anyone questioning that independence, and I think this explains the balkanization of SRE teams. Without this gift of autonomy, teams become much less self-motivated; with it, the teams all share a culture of fierce creativity, drive, and independence, but they also become a bit sectarian. SRE teams as a whole are "progressive" in that they are constantly tearing down and improving technology, but very "conservative" within their respective teams about custom tools and processes. *This is the Curse of Autonomy: disparate and stubbornly "conservative" teams.*

We can reasonably infer these consequences of team autonomy, but the preceding account is hardly evidence. We also can't really use introspection and surveys as evidence, because these effects of autonomy seem to be unconscious. However, an example of the Curse of Autonomy elsewhere—especially outside of Google—would be good supporting evidence.

The SAS is one such well-documented example.

Evidence Outside of SRE: The SAS

The Special Air Service (SAS) is the British Army's special forces, but what makes it special isn't obvious.

The SAS and the British regular army do the same things: jump out of airplanes; attack enemy positions in jungles, deserts, and cities; and learn how to use many kinds of weapons. It's all the same stuff.

The difference between the two types of forces is the gift of autonomy. In the regular army, almost everything is planned for a soldier—how they train, what they wear, what rations they pack, how many bullets they have, where they'll be and when, and so on.

SAS team autonomy

In this respect, the SAS is the reverse of the regular army: every soldier and team has the gift of autonomy. Higher-ups decide *what* a mission is, but SAS soldiers train, plan, and execute the *how* all on their own. They decide how they train for the mission, what arms they use, what they wear and eat, and so on.

Autonomy is a focus of the three-month admission test, called *Selection*. It is the reason why there is no yelling or other verbal abuse. For example, every day and all day for the first month of Selection, candidates run alone across featureless, hilly terrain with heavy packs, navigating their way from one checkpoint to the next, using only a map and compass.

At each checkpoint, a lone officer impassively asks the candidate three questions. The officer wants the candidate to show on their map where they think they came from, where they think they are now, and where they think they have to go next. Get anything wrong, and they've failed Selection. Answer correctly, and the officer will quietly nod them on their way with a few noncommittal words. There is no "well done" or "hurry up," or drama, or any feedback of any kind about how they are doing. The candidate never knows whether they are making good time or not.

This deadpan quietness is by design. It means that applicants who need approval or urging to keep up the pace don't receive it and drop out voluntarily before the month is over. The SAS doesn't just want endurance athletes: it wants soldiers who, if given autonomy, won't need any additional prodding or encouragement to use it.

SAS team differences

If I am correct about the Curse of Autonomy, we should expect the SAS to consist of small teams, each of which performs the same soldierly tasks but using radically different approaches and techniques. And that is, in fact, precisely what we do find.

In any given year, the SAS has a single regiment of about 500 soldiers. The regiment is divided into squadrons, which are divided into troops. A *troop* is a small team of about 10 to 15 soldiers, the same size as an SRE team.

SAS

In the excerpt that follows from an SAS soldier's memoir, he describes his first day after he joined his first troop. A *target* here is a cardboard enemy soldier, and detecting it is called making a *contact*. The other soldier, Colin, was a veteran in the new troop, which was stationed in a jungle.

> We were going to do jungle lanes, very much as [I'd] done on Selection. We patrolled along in a group of two...practicing contact drills.
>
> As Colin and I were patrolling, we saw a target. I remembered my [Selection training] drills well; I got some rounds down, turned, and ran back. Inexplicably, Colin gave it a full magazine, dropped in another one, and kept going forward.
>
> He turned and shouted: "What the #@%! are you doing?"
>
> "We weren't taught to do it like that."
>
> "Oh for #@%!'s sake."
>
> Every squadron did it differently, I discovered, *and so did every troop*.[6]

In other words, each small team of 10 to 15 SAS soldiers, granted full autonomy to design its techniques and training, has its own idiosyncratic way of doing fundamental tasks, such as how to respond to running into enemy soldiers on a jungle path. And like SRE teams, SAS troops don't criticize one another for doing it "wrong."

If that's the case in the SAS, where doing something the "wrong" way can get you and a lot of other people killed very quickly, we shouldn't be surprised to find SRE teams defending their own singular approaches just as jealously. It's the Curse of Autonomy.

For any release automation tool that aimed for adoption by many SRE teams, the consequences of the Curse of Autonomy were serious. Any tool would need to find a way to overcome these sectarian barriers: the barrier of every SRE team doing similar things very differently, and the barrier of SREs being prickly about being told how to do things by anyone outside of their team.

Only one release automation tool was designed for and well adapted to this environment. That tool was Sisyphus.

---

6 Andy McNab, *Immediate Action* (New York: Dell Publishing, 1995), 174.

Overcoming the Challenges

In the fall of 2008, as soon as Cory returned empty-handed from the ReleaseItNow team, the immediate task was to stop the bleeding: automate Gryphon, in any way they could. Over two weeks, a few Search SREs accomplished this by extending their load testing tool into a rollout wrapper script. Now, if all went well, a Gryphon cluster rollout could be completed with a single CLI command and take two to three hours. If something went wrong, SREs could restart the script where it left off. The script could even do drains by calling a Traffic backend. If something went wrong—for example, if a load test failed—the script sent an email. SREs no longer needed to stare continuously at graphs. Instead, the script scraped and compared the graphs to detect failures.

Cory wanted to stop there; he felt the script was enough. But other SREs wanted more. There were just too many things that the script couldn't do. For example, you couldn't schedule the script to run at an opportune time. Although it might take 45 minutes to perform a certain work task, an SRE might need to wait for hours, checking for when there was enough capacity to do it. Sometimes this meant pushing at night.

Search SRE convinced Cory that it needed a general tool, not just a script. The problem was that the team knew writing such a tool would take a great deal of engineering hours. So from the outset, it decided that it would build this tool—Sisyphus—not just for Search SRE, but for all of SRE. The team had three reasons:

- If a manager up the chain ever challenged the team over spending so much time on a single Search SRE tool, it could justify the work by saying Sisyphus was for all of Google.
- The team needed to justify the extra work to itself, too. Sisyphus wouldn't be a good use of its time if the result wasn't something all SREs could use.
- If anyone demanded that the team stop because ReleaseItNow already existed, it could truthfully counter that unlike ReleaseItNow, Sisyphus would cover all the SRE use cases ReleaseItNow didn't, which was most of them.

We already know the ending to this story: Sisyphus was widely adopted by Google SRE despite the many challenges it faced. In the

following sections, we examine the (often incidental) factors that contributed to its success and offer actionable takeaways.

Code Choice

When Sisyphus was being written, any tool that wanted to appeal widely to SREs would need to be written in Python, because of the Red Queen. SREs often must fix or modify their tools on the fly. For this, Python is ideal, which is why it was then the SRE language of choice. Compiling, running, and testing code changes takes much longer in C++ or Java than in Python. This was even more true 10 years ago than it is today, and language difference significantly contributed to a reasonable mean time to repair (MTTR) when an incident occurred.

Through a combination of luck and technical decisions, unlike some other SRE release tools, Sisyphus was written in Python.

Takeaway

Other tools and approaches might not have taken the Red Queen into account as Sisyphus did, perhaps relying more on asking teams to switch to a new, common tool for the greater good. Sisyphus instead adapted to the Red Queen environment by using a programming language that best suited that environment.

When choosing a tool or its language, you might find it useful to take the environment into account in a similar way, especially if you have no control over that environment. For example, it can be helpful to consider how a tool's language fits into the existing culture and circumstances. What are the specific needs of your audience? Does the majority of your audience need to be able to modify the code? Is quickly modifying the code important? Will the people most likely to be working with the code know the language you're using? Does the toolchain support the language, minimizing the compilation/configuration/build/deployment workflow?

Adaptability

Both the Red Queen and the Curse of Autonomy meant a universal release tool couldn't be a my-way-or-the-highway program. For example, ReleaseItNow required you to do releases a certain way. If you didn't or wouldn't do releases that way—if, for example, you

couldn't have old processes running alongside new ones—you couldn't use ReleaseItNow.

Because of the Red Queen and the Curse of Autonomy, no SRE team had the time or inclination to redesign their processes around some new automation tool, no matter how wonderful it allegedly was. However, although no SRE team would adapt to a tool to adopt it, SRE teams might adopt a tool if that tool could be adapted to the team and the team's existing processes.

In other words, a team wouldn't adapt to adopt, but a team *would* adopt something that *could* be adapted.

Sisyphus's plug-in architecture fit this requirement. Sisyphus was really just a scheduler that ran other programs. It could run anything that could be tailored as a Sisyphus plug-in. If a team wanted to automate a custom CLI, it was easy to write a Sisyphus plug-in wrapper around it in Python. Then, Sisyphus could call that CLI, and the team could tell Sisyphus when to call the CLI—over and over again. Sisyphus could automate anything!

Again, by some combination of luck and ideas based upon the problems Cory encountered on the ReleaseItNow development project, this design also met the requirements of the Red Queen and the Curse of Autonomy. Sisyphus was actually designed around plug-ins so that an SRE team wouldn't need to redeploy and reboot all of Sisyphus every time it wanted to make a small rollout change. With a plug-in architecture, you could leave Sisyphus and dozens of plug-ins running, and just redeploy the one plug-in you needed to change.

Takeaway

When engineers are strongly motivated by something they value, we can think of that value as their *engineering culture* (or part of it). Google SREs are strongly motivated by the gift of autonomy Google gives them. Other engineers at other companies and in other organizations might have other cultures.

Teams might not be enthusiastic about adapting their already existing culture to a new tool. Instead, maybe that tool can be adapted to the team and its culture. In Sisyphus's case, adapting to SRE culture meant accounting for teams' entrenched and disparate processes. Plug-ins provided that adaptability. Other tooling-specific avenues

might include "zero configuration" protocols that discover and roll out systems that conform to a certain shape, or configuration as code, which uses code snippets to drive system behavior.

Adoption

We know very little about how Sisyphus actually spread early on. When I interviewed Cory and others 10 years later, most couldn't remember much about how adoption spread. Except for all the related technology, which these engineers could still remember in searing detail, that part of the story was mostly a blur. Neither Cory nor anyone else recalls having much of an adoption plan or strategy.

Cory and others do remember that the first non-Search SRE team he approached, in March 2009, was Traffic SRE. Traffic dreaded its excruciating procedure for rebooting frontend servers. In just two hours, Cory built a Sisyphus instance and UI to perform this task with just a few clicks. As SREs disliked creating UIs, a tool that had a UI—with plug-ins, to boot—seemed somewhat magical.

Some teams heard about Sisyphus through the grapevine and began using it on their own. Cory approached other teams directly. He began adding important features to make it ever easier for teams to use Sisyphus to automate more of their many different processes. These features included the following:

- A big red "panic" button to make Sisyphus stop
- The ability to "branch" among alternative steps instead of using fixed steps
- The ability to leave comments on steps, which came in handy for SREs handing off an unfinished rollout
- Email alerts, which were essential for extremely long rollouts
- The ability to handle dependency chains, making Sisyphus smart enough to push dependent binaries first
- Data scraped from the monitoring system, which you could email or diff against previous versions of scraped data
- Dynamic links in the Sisyphus UI to external monitoring graphs

Cory sat with SREs from other teams for hours, watching as they conducted a long rollout, to discover new aspects to automate. He

was taken aback by how much manual toil SREs were often willing to put up with.

One by one, Cory overcame the different roadblocks to adoption. For example, a team's SREs might object to using Sisyphus by saying, "Sisyphus won't canary for us," or, "Sisyphus won't do a capacity check before draining." In response, a Search SRE constructed a checklist analogous to a pilot's pretakeoff checklist. He went to each member of the Search SRE team and asked, "What do you do before you redirect traffic to a different cluster?" Every single SRE had a different answer, based on something particular that had gone wrong for them once during this procedure. Cory then coded all these answers as checks into a single Sisyphus plug-in. As a result, Sisyphus could test far more drain conditions than any individual SRE did.

So now, when a team claimed that Sisyphus couldn't perform some check, Cory could show them that Sisyphus *could* do that check. And if Sisyphus couldn't, Cory would either code a Sisyphus fix or point out workarounds. This cycle repeated until teams no longer had reasonable objections to using Sisyphus.

Cory didn't work alone on adoption. He received both logistical *and* moral support from Search SRE. For example, after a detailed and frustrated lament to his manager about how he just couldn't get some team to use Sisyphus, the manager said, "Then don't." In these cases, Cory needed an extra push to stop trying so that he could focus on other teams instead.

Takeaway

If you want your organization to adopt your software, it might be helpful to get to know your customers as well as you know your own code, if not better. It can sometimes help adoption to spend time observing and working with your customers and tending to their particular needs.

Scaling Adoption Efforts

Even though one-on-one support was a great way to encourage SRE teams to adopt Sisyphus, scaling adoption became a lot easier after other Search SREs and a technical writer created some basic documentation and tutorials.

Because Google SRE didn't have any mechanisms or institutional support for staffing a cross-team tooling project, Google SRE software development functioned tacitly a bit like open source software: if you use our tool, we strongly encourage you to contribute code and bug fixes to Sisyphus code in return, but we can't force you to. If you can't contribute, maybe someone else on your team can. Perhaps unconsciously, Cory moved even further toward the open source model. Instead of just becoming the owner of Sisyphus code, he and the Search SRE team became owners of the community that used Sisyphus by being attentive to its needs, often even anticipating them. This meant mixing socially with other SRE teams to both become familiar with their release processes and to gain their trust.

Takeaway

Some tactics for scaling adoptions are universal, whereas others will be specific to your organization and its culture. Documentation and written tutorials can enable people to adopt your software without intensive one-on-one training. Within Google, we found that we could sustain Sisyphus's development by appealing to other aspects of SRE culture—relying on open source dynamics to motivate code contributions.

Adapting to SRE Psychology

Convincing some teams to adopt Sisyphus turned out to require social and diplomatic skills Google doesn't look for in its engineer hiring interviews.

When Cory showed the Traffic SRE team his Sisyphus solution to automate its manual GFE restart process, he found out the hard way that one of the engineers had been working for some time on a tool to automate it. Now, Cory was demonstrating a Sisyphus solution that took him only two hours to code up—directly in front of that engineer and all his SRE teammates. He learned an important lesson from this awkward interaction: from then on, he wouldn't just study teams' rollouts in depth; he would always do his people-centric homework, too, and adapt his pitch as needed.

It turned out that all too often, a team's reasons for resisting Sisyphus weren't technical at all. After being shown that Sisyphus could do everything they wanted, some teams would switch to excuses. "Oh well," they'd say, "We don't have time this quarter." Or, "There's

no easy way to add what we need to Sisyphus; that would take a super long time and what we have already works." And so on; always the same old song of Sisyphus.

These excuses were unconvincing because Sisyphus would clearly lighten the team's workload.

Automate your way up one level

To provide a typical example, after a number of weeks and beers, one person revealed the real reason he didn't want to use Sisyphus. The team's SREs were comfortable with their manual solution and felt threatened by Sisyphus: what would become of their jobs if they implemented it? Wouldn't the tool just replace them? This turned out to be a fairly common reason for resistance throughout SRE. Tunneling down to this root objection always took a long time and a lot of socializing.

One SRE came up with an honest argument that overcame these fears. One motto in SRE is, "Automate your way out of a job." It's supposed to mean "if your job involves toil, write code to automate the toil away." Instead, the SRE suggested that a better way to think about the motto was, "Automate your way up a level." This meant two things. First, it meant promotion—automating your way up into a higher-skills job.

More important, it meant working your way up one abstraction level. For example, for years SREs ssh'ed (remotely logged in) to each individual machine. Then, a tool that could run a command across multiple machines came along. With the advent of Google's cluster manager, Borg, the individual machines were abstracted away, too. A new Borg tool brought the level of abstraction to the Borg process, mostly ignoring individual machines. Then ReleaseItNow and Sisyphus could manage processes across multiple clusters. People didn't become redundant at any of these levels. Every abstraction level just yielded different work. Each level just made SREs more productive, because the same number of SREs did more work. And then customer demand always surged to create more work that eventually needed automating.

So Sisyphus wouldn't put any SREs out of a job. It would just change their jobs, and make their work more rewarding.

Not automatable

In another example, one team vehemently insisted that it had already tried three times to automate its processes and that those processes simply couldn't be automated—not by Sisyphus, or anything else. Conversations went on for months. Eventually, Cory said to one engineer he had come to know well, "Look—I believe you. I believe that your current process can't be automated. But maybe the process could be changed, maybe simplified, and then it could be automated with Sisyphus?"

Remember: SREs are sensitive about being told how to do their job—unless you get to know them well first, and that takes a lot of time. Telling an SRE that "maybe" the hundreds of lines of code they wrote could be simplified is tacitly criticizing their solution for a process they know best. Socially, it's challenging to pull off.

But persistent and thoughtful conversations did eventually win out. This approach was slow, but it worked. Two weeks later, that engineer had simplified the team's process, and soon thereafter the team automated the process with Sisyphus.

Takeaway

Cory and his team needed thick skin, patience, and sensitivity not to give up, no matter how many times they heard: "Thanks, but no thanks." The accompanying reasons didn't always make sense. Their adoption approach was slow, but it was effective. It took a long time to earn other engineers' trust so that they confided their real reasons for not wanting to use Sisyphus. It took more time to find ways to address those concerns honestly and compassionately. If other adoption approaches are not working for your organization, the Sisyphus team's approach might help.

Passable, Not Perfect

Anecdotal evidence and my own personal experience suggest the overarching reason a team adopts a tool at Google is word of mouth. The Sisyphus team simply formalized this approach. They actively set out to become that "other SRE" whose technology choices and recommendations you trust. There was nothing devious about this approach—that trust was won fair and square, without deception. Although Sisyphus may not have been the prettiest or most theoretically elegant tool, it became the most popular.

And because of the Red Queen, it didn't need to be the best tool; it just needed to be passably good. The staffing crunch meant that the barrier to entry and the speed with which an automation tool could be built out was more important than elegance of design, scalability, or maintainability. If Tool A can do the job well enough now, if the work needed to cope with its shortcomings is less than the work needed to learn and build on Tool B, Tool A is preferred. Sisyphus's success followed that model. For example, a Sisyphus instance would run out of RAM and crash if it had too many tasks, but there was a crude and easily usable workaround: just run many more Sisyphus instances. Other automation tools had worker pools to avoid this problem, or more live object statuses—better, more elegant designs. They just weren't as well adapted to overcome the Red Queen and the Curse of Autonomy.

Takeaway

In some circumstances, engineers might prefer to build a nimble and passably good tool that can overcome barriers to entry rather than perfecting a tool that might or might not become widely adopted. This approach was first proposed many years ago as an engineering pattern called *Worse is Better*. Sisyphus's use of this pattern suggests why it can be useful in other organizations facing adoption challenges.

Success and Its Costs

In the mid and late 2000s, there were three significant barriers to SRE-wide adoption of a practice or tool. The Red Queen and the Curse of Autonomy were the first two barriers. The fact that SRE was not yet set up to support development of cross-team tools was the third. There was no central budget, program, or authority that a project like Sisyphus could apply to for additional engineering staff and backing.

Success

Despite the three strikes against it, Sisyphus achieved some measure of standardization across SRE and beyond: it became one automation tool that everyone did use in common, for more than just rollouts. The disparate processes of individual teams for similar tasks converged enough so that Sisyphus could automate them.

Sisyphus's adoption success can be explained by how well adapted the code's design choices and evangelizing efforts were to overcoming the three challenges. Some of this adaptation was sheer luck. Python and a plug-in design were partially chosen for technical reasons, but not consciously chosen to address the Red Queen or the Curse of Autonomy problems. They just happened to address these problems very well.

The fact that some of Sisyphus's engineers had the requisite patience and diplomatic skills to pull this off was also luck. It takes the thick skin and empathy of a sales rep to hear "No, No, No" and to continue trying diplomatically. Most engineers don't have this ability, and Google doesn't interview for it.

Costs

Part of the history of Sisyphus is how it was adopted; the consequences of widespread adoption is another part. On the plus side, Sisyphus obviously removed a lot of manual and error-prone labor. Sisyphus automates a great deal of SRE and other engineering work.

But there were also minuses.

Sisyphus came with heavy technical debt that has never been paid off. Sisyphus has all the problems of any large Python project, which has made it excruciating to maintain. As late as 2016, a single bug could take eight releases to fix, because the lack of type safety made it impossible to discover all uses of an object or method through unit tests or regex. The flexibility that overcame the Red Queen became Sisyphus's greatest weakness.

Sisyphus exacerbated the consequences of the Curse of Autonomy in SRE. Although teams were nudged to "simplify" their practices to make them automatable with Sisyphus, "simplify" really meant "Sisyphy." As soon as Sisyphus had automated a team's release processes, there was no more pressure to streamline those processes further or standardize them across teams. Teams also kept all their idiosyncratic tools because they could automate them with a simple Sisyphus plug-in wrapper.

To scale up, SRE teams might eventually need to standardize either their release processes, their tools, or both. If so, Sisyphus helped postpone that day of reckoning by almost a decade.

If Sisyphus showed how to get software adopted across SRE, perhaps it also demonstrated that adoption is not a reliable signal of code quality or fit.

Although the Sisyphus rollout tool is far from perfect, and the story of its adoption is full of luck, coincidence, and frustrations, its history might suggest some lessons that any organization can apply to reduce the pain of similar pursuits.

An Actionable Takeaway

For engineers to have a "culture" is for them to be motivated by something they value, rather than being ordered around. For example, having a "postmortem culture" means that engineers write postmortems because they see their value, not because the postmortem is required. At Google, SREs are strongly motivated to work hard, and in an independent and creative manner because of the gift of autonomy that Google gives them: this culture of autonomy motivates driven, independent behavior.

Organizations sometimes try to change engineers' behavior by changing their culture, but changing culture is very difficult. This case study about Sisyphus describes how an organization changed its SRE's behavior without trying to change their culture. Every SRE team was using its own unique custom tools: this was the behavior Google wanted to change.

Sisyphus altered much of that behavior without attempting to overcome either SRE culture or circumstances. Instead, Sisyphus adapted to the SRE culture of autonomy and to the hectic environment SREs worked in. Through its use of plug-ins and diplomacy, Sisyphus addressed the Curse of Autonomy, and through its use of Python it addressed the state of the world, the Red Queen. This appears to be why Sisyphus was almost universally adopted by Google engineering teams, despite the long odds created by its lack of staffing, support, and dedicated, chaos-free development time, and its use of suboptimal source code.

Culture can be changed, but doing so is challenging. Starting in 1973, it took the US Army a decade to create a culture of doing post-

mortems, which it called After Action Reviews (AARs).[7] Most industries don't have that kind of time. Could the army have gone faster by adapting to existing army culture to get AAR behavior? We don't know.

What we do know is that Sisyphus has shown us another way, one that can sometimes be faster *and* easier.

Perhaps your organization is trying to change behavior by changing culture. For example, perhaps you're trying to get engineers to gate all submits on code reviews by peers instead of allowing engineers to submit code whenever they feel like it. You might be trying to inculcate a code review culture by getting engineers to see the value of code reviews and therefore to be motivated by the value of code reviews.

This strategy can work. But if it is proving very difficult, we hope the story of Sisyphus might encourage you to experiment with a different approach.

First, figure out exactly what the desired behavior is—for example, that all engineers have their code peer reviewed. Second, ask yourself if there is a way to adapt your tools and processes to some aspect of your engineers' culture—of the values that already motivate them—and to the limitations imposed by time and circumstances. Whether such an adaptation is possible, and what it will look like, will be different for different organizations.

We hope that the history of Sisyphus will inspire you: that the story inspires you to be alert to these possibilities rather than to always choose to change behavior by changing culture.

It certainly inspired Google to.

7 G. R. Sullivan and M. V. Harper, *Hope Is Not a Method* (New York: Random House, 1996), 192.

About the Author

Richard Bondi has been an engineer at Google since 2011, specializing in the entire web stack and working on travel applications. In 2016 he converted to SRE, and then joined the SRE tech writer team. Before Google, and after leaving his political philosophy PhD program to join the first of many internet startups, he wrote a book on the Microsoft CryptoAPI, published by John Wiley & Sons, Inc.