The Song of Sisyphus*

How one automation tool achieved widespread adoption by adapting to Google diversity.

* For SRECon19, this talk was renamed The Curse of Autonomy.
Who am I?
- I'm Richard Bondi, been at Google since 2010, was a production dev in Travel and never learned much about SRE. Three years ago I did a rotation in SRE and, partly because of the awesome onboarding experience, decided to stay. The other reason was they said they'd pay me to write docs, which I always wanted to do. So I'm one of the few SRE techwriters, but they still have me write code.

What's this about?
- It's a summary of a 10'000 word history I researched and wrote of something unusual: how one rollout tool came to be adopted by virtually all SRE teams and many dev teams inside Google. So I'll have to leave a lot out. The history is supposed to be published externally, maybe this year, but I'm not in that loop currently.

What is Sisyphus?
- Sisyphus is the tool. I have a screenshot coming up.

Why does it matter?
- Google has a hard time getting SRE teams to standardize on tools and processes. Once your SRE org reaches a certain size, two things happen: standardization starts to matter, and you discover it's hard. My idea was: can we learn anything from when it happens organically? So I got the OK to find out, interviewed about a dozen people, did a lot of code archeology, and wrote the history.
The Mystery
1. Sisyphus screenshot
2. Sisyphus alternatives
3. Adoption graph
4. Mysteries
5. Overview of rest of talk

So in the second part of this talk, I'll explain why there's a mystery behind Sisyphus' adoption. In the next few slides I'll show:
- a Sisyphus screenshot to explain what it does
- a list of alternatives also written in-house, which didn't catch on
- one of many graphs showing that I'm not making this up, Sisyphus really was adopted like crazy
- and finally, why this crazy adoption is mysterious.
- Then I'll give an overview of the rest of the talk, which will solve the mystery.
Sisyphus is essentially just a task scheduler. Each task can execute a bunch of CLI commands that a human would have to do manually. You can tell Sisyphus to proceed based on some responses, or to stop until a human presses a button.

In the screenshot, it looks like a binary candidate was built for QA, and then Sisyphus stopped to ask if it should proceed to canary the binary. If the human agrees, Sisyphus will carry out a bunch of canary steps that would be toil for a human.
Sisyphus alternatives: an incomplete list

1. Code name
2. Code name
3. Code name
4. Code name
5. Code name
6. Code name
7. Code name
8. Code name
9. Code name
10. Code name
11. Code name
12. Code name
13. Code name
14. Code name
15. Code name
16. Code name

Around the same time Sisyphus was created, many other SRE teams wrote their own rollout automation tool to do similar things.

I can't name the tools, because they are internal code names. But you get the idea: there were a lot of them. An SRE team would build their own tool, but the tool would never spread to other teams. Only Sisyphus spread.

And that's not even the mystery: the mystery is even deeper, as we'll soon see.
Sisyphus is weird in a way that made it easy to track whether it was being used. It's not a shared service. Instead, you subclass the source, code your own additions and tasks, build your own binary, and then push it into prod and run it yourself. This means that an SRE team is always committing code to their Sisyphus throughout the year, because rollout processes are always changing. If the commits stop, it means the team stopped using Sisyphus.

In the graph:

They yellow line shows commits to all these Sisyphus binaries: as the years go by, there are more and more commits, which strongly suggests more and more teams have spun up their own Sisyphus.

The blue bars show that despite this, the ratio of commits to unique authors isn't changing. So the commits aren't due to a few heros or a few teams.

This and other data strongly suggest what every Google SRE already believes: that Sisyphus is everywhere. Not only are new teams adopting it, but when they do, they continue to use it.

Sisyphus is an adoption success story.
Where “success” is narrowly defined as success of adoption, Sisyphus's success is baffling for three reasons.

1. Whatever explains Sisyphus’ adoption success, it cannot be best practices of software development: Sisyphus violated them all. Sisyphus had no roadmap, no schedule, no charter, no PRD, no SLO, no defined development process, no clear team, no product manager, no project manager. The design doc was written after its first version went operational, not before.

2. Code quality can't explain Sisyphus' success: even the original authors say it's bad. One said to me: “I would never use Python now, I think it is the worst thing to use [for large projects].” Fixing Sisyphus bugs is hell.

3. Lastly, for all of its nine years, Sisyphus had the opposite of SRE leadership support: leadership were always trying to get rid of Sisyphus. They had perfectly good reasons. For example, Search SRE managers were short of SREs, and got mad because their few SREs were writing code used by the rest of SRE — not fair! So they tried to replace Sisyphus with something else that had its own team, and would be better quality. One Sisyphus engineer wrote on his whiteboard: "Sisyphus will be replaced by," and the name of the latest proposed replacement. He kept crossing the replacement out and writing the latest one underneath it. The list grew towards the floor. We've no picture, because this was 2008-10.
The rest of this talk is about why I think Sisyphus succeeded despite these baffling odds, over all its rivals.

The explanation is not merely my opinion. In the 10,000 word history, I assembled a lot of evidence for my explanation. Here, I'll have to just sketch the evidence.

Notice that the problem is (software) adoption, but the solution was adaptation — adapting to SRE culture.

The problem seemed to be how to change SRE culture to get it to adopt a single tool.

The solution Sisyphus found, really by accident, turned out to be to adapt to that culture, not to try to change it.

The rest of this talk has the following structure: I'll describe two big obstacles that *any* rollout tool faced if it wanted to be widely adopted. I call these The Red Queen and The Curse of Autonomy.

Finally, I'll describe how only Sisyphus overcame these two obstacles, of all its rivals. And that will be my explanation of the mystery.
Red Queen
- *Alice’s Adventures in Wonderland* is a 19th century children’s book that is famous mainly in English-speaking countries for its imaginative paradoxes of language and logic. For example, in its sequel *Through the Looking Glass*, a chess piece named the Red Queen has to run as fast as she can just stay in one place. The Red Queen has become a metaphor, both in science and literature, and as such it also captures what being an SRE was like in the years 2007 and 2008.

Pyramid Scheme
- In 2007, there was a cynical joke that SRE was a Pyramid scheme: you got hired thinking you’d do interesting work, and instead it was toil for a year until you could get allowed to be on-call, when *maybe* you’d be allotted time to do development work.

Search SRE and Munchkin
- Rollout of a version could take weeks of back-to-back work, because each rollout was so huge, and had to do dozens of them to be completely rolled out to the planet.
- Lots of toil, but very demanding, highly-skilled toil: e.g. interpreting dashboards correctly, knowing how to rollback from any point, etc.

Example of just one aspect of rollout: Load testing
- 2x for cascading
- Cache testing of different kinds of graph (node vs leaf) failures, _and_ restoration/refresh of the cache when back up
- It was very stressful, because failures had to be detected by humans staring at
graphs, possibly for hours, while getting the zone up as quickly as possible.

ReleasetItNow
- Three non-SRE engineers decide to solve rollouts automation for 80% of Google: specifically, be able to do multiple zones without intervention, not just one at a time. Ed Pizzi from Search SRE decides to go help them for a few months, to make sure RIN can do what Search SRE needs.
- Found RIN had excellent design, but too many assumptions that showed unfamiliarity with SRE work. RIN made assumptions that were reasonable for many releases at Google, but less so for the complex releases that required SRE teams. It assumed that pushing a binary to a single zone would take about an hour. It assumed that testing was simple and quick; that the order in which you pushed binaries didn't matter; had no load testing step; and so on. Design too hard to refactor: couldn't easily add add branching to a rollout sequence, drains, load testing.

Red Queen with hair on fire
- So end of 2008, Ed Pizzi returns to Search SRE empty-handed. RIN released, but not usable by Search SRE or other big SRE teams.
- End of 2008, workload doubled, but number of SREs didn't. Nor did the number of zones. Munchkin plus a whole other new thing that required as much rollout effort as Munchkin did.
- The implications were very serious for _any_ rollout tool that wanted to be adoptable by _all_ SRE teams. Before, just had to be useful. Now, had to be useful under immense pressure of the Red Queen environment — a Red Queen with her hair on fire.

We'll now leave our protagonists Ed Pizzi and the Search SRE team for a while: we'll return to their story later.
Paradox
- While SREs share a common culture (can-do, driven), different teams do similar work in radically different ways — and are prickly about it.
- SREs know this. One SRE told me his move to a new team was like a Star Trek Mirror Universe. Everything the same, but done differently.
- Example: Search SRE and Traffic SRE split from the same team, but evolved differently. Search pushes, then commits the config change. Traffic is always other way around: commit, then push. Search thinks that's almost sacrilege.
- Quote: "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."

I'm going to argue that SRE autonomy explains this diversity.

First, examples of level of autonomy in Google SRE (and some dev teams too).

Personal evidence:
- SRE team I rotated into, was like Larry and Sergey had showed up day after team was hired and said: "Here are your keys. Obviously all of Google depends on this stuff, so figure it out for us, and keep it working. Microkitchen is down the hall to the left. Thanks, and welcome to Google!"
- Team set OKRs, managers hustling to keep up — not the other way around. (That's a slight exaggeration to illustrate how driven the SREs were: the managers actually were and still are awesome.)
Other evidence:
- Postmortem by SRE team on why joint project with a Prod Dev team didn't work out. The Prod Dev team invited SRE team to join, then came a sudden divorce. Three factors identified. Main one: autonomy. On SRE, engs made design decisions; on Prod Dev, Prod Managers made them (e.g. for features in product). What would happen:

-- A dev engineer would be outraged to discover that an SRE eng had changed the design, without discussion with the SRE or Prod team's manager.
-- The reverse happened too: an SRE eng would agree on a design with a Prod engineer — then be outraged to learn 2 weeks later that the Prod manager had changed the prod engineer's design, and without involving an SRE engineer.

To the Prod Dev team, the SRE team seemed off the reservation: making design changes without managers. So, the Prod Dev team pulled the plug. But actually, the SRE postmortem found, the problem was different cultures, not SRE disorganization.

So, that gives you a picture of SRE autonomy. It's a gift of autonomy that Google gives its SREs.

What I argued was that the gift came with a curse: a sort of tribalism. I argued that this autonomy explained why teams had such different tools and processes for similar things like rollouts.

- My initial theory was just speculation: autonomy creates team differences: jealously guarded, don't tell me how to do my job, now that I've been gifted with autonomy to do it.
- That's a nice theory. But what evidence did I have? Ideally, could I find evidence of the same thing outside of Google?
- And I did: the SAS. They are the British Army's special forces, but it's not obvious how they're special or different from the rest of the army. They do the same things: shoot guns, jump out of planes, march around, etc.

— The difference is autonomy. For example, during first month of selection, just running in featureless, hilly part of Wales with a map and a heavy pack. Every few miles the soldier encounters an officer with three questions: Show my on your map:\n- 1) Where you think you came from?
- 2) Where you think you are now?
- 3) Where you think you're going next?

If the soldier gets anything wrong, it's over: you're out ("binned").

If the soldier gets them right, the officer just nods him on his way. There is zero
feedback about how well or badly you're doing. The SAS wants not just endurance athletes, but people who, once given the gift of autonomy, won't need any yelling or praise to actually use it.

The SAS structure is about 500 soldiers in a regiment, split into squadrons, split into troops. A troop is about the size of an SRE team: about 10.

If I was right about autonomy, we'd expect SAS troops to do similar things very differently, and be prickly about it.
And that is in fact exactly what we do find.

This is illustrated in an anecdote from a memoir of a soldier named Andy, on his second day in his first troop. Andy was paired with a veteran soldier named Colin, and Andy was excited to show Colin how well he had learned these drills during his three month selection courses. A contact popped up down the trail. And knelt down, fired three careful shots, then ran back up the trail. At the same time, Colin unloaded his magazine at the target, then continued walking towards the target while loading a new magazine. Colin turned around to ask: "What the f*** are you doing?" Andy, shocked, said: "I wasn't trained to do it that way!" Colin replied with an expletive. Andy soon discovered that "every squadron did it differently, I discovered, and so did every troop." There's no such thing as "the SAS way": every troop is different. Just like every SRE team is different.

So the curse of autonomy is that every team tends to do things their own way. The blessing of autonomy is how much drive and energy it inspires. The two are like yin and yang. The challenge is to reduce the negatives of the curse without destroying the blessing.

And Sisyphus managed to pull off that trick.
The reasons Sisyphus adoption was so successful:

Python: given the Red Queen, any tool trying to be a common rollout tool would have to have been written in Python. The only alternatives were Java, C++, and bash: too slow to compile back then, too complicated, not hacky enough. There were Sisyphus competitors written in those languages.

Plugins: for totally unrelated reasons, Sisyphus chose a plugin architecture. Got your own custom CLI that you want to automate? No problem: write a thin Sisyphus plugin wrapper around it in Python, and you're good to go. This meant teams didn't have to adapt their tools and processes to Sisyphus, they could adapt Sisyphus to how they worked. Usually painlessly. With other tools, it was their way or the highway.

Passable: Sisyphus didn't work well, but well enough. For example, it had no thread pool; so instead, SREs would just spin up more instances to get around the memory maximum. Dirty and toily, but good enough, given the Red Queen.

Psychology: often, the reason teams gave for not using Sisyphus weren't actually their reasons. Just one example: after a few weeks, one SRE admitted the real reason his team didn't want it: they thought it would automate them out of their jobs, literally.

The Search SREs worked up a different pitch that sold the team. "Automate your way out of a job" is an unfortunate slogan, it should really be "automate your way up a
level of abstraction":

For years SREs ssh’d to each individual machine. Then Google invented an internal tool to allowed easy ssh to many machines at once. Then came Borg (Kubernetes’ predecessor), and the individual machines were abstracted away, too. Then a Borg tool abstracted away the notion of an instance, replacing it with "job." Now, Sisyphus could manage jobs across multiple Borg datacenters. What’s next: abstracting to continents? The planet?

Because the Internet is so great, demand always surges to fill demand. When SREs automate away toil and become more productive, it's just a matter of time before they have to again: demand swells until you've got toil again. Automation will never put SREs out of work, at least not in this lifetime.
Questions?

rbondi@google.com