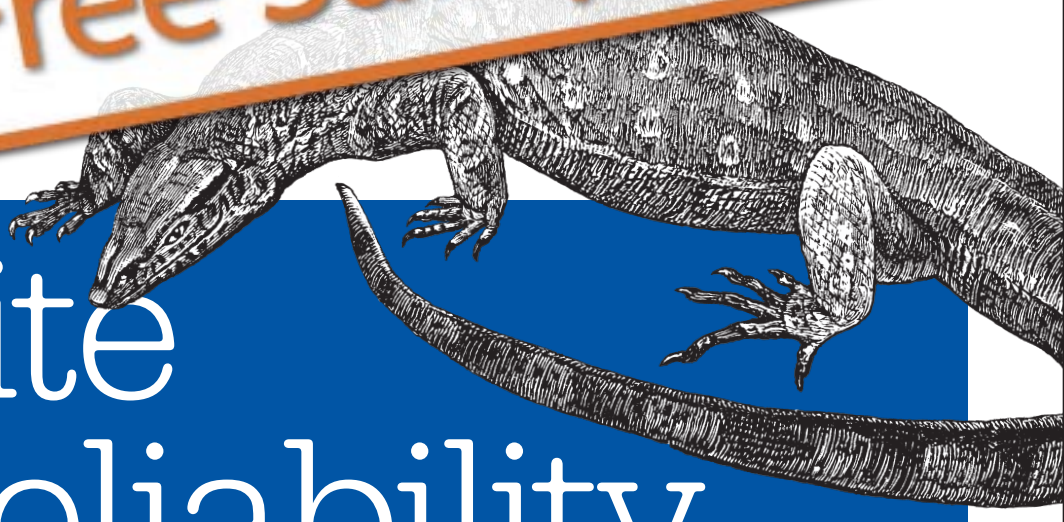
O'REILLY®

Free Sampler



Site Reliability Engineering

HOW GOOGLE RUNS PRODUCTION SYSTEMS

Edited by Betsy Beyer, Chris Jones,
Jennifer Petoff & Niall Richard Murphy

Site Reliability Engineering
Edited by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: Kristen Brown
Copyeditor: Kim Cofer
Proofreader: Rachel Monaghan
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2016: First Edition

Revision History for the First Edition
2016-03-21: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491929124* for release details.

978-1-491-92912-4

[LSI]

CHAPTER 7

The Evolution of Automation at Google

Written by Niall Murphy with John Looney and Michael Kacirek
Edited by Betsy Beyer

Besides black art, there is only automation and mechanization.
—Federico García Lorca (1898–1936), Spanish poet and playwright

For SRE, automation is a force multiplier, not a panacea. Of course, just *multiplying* force does not naturally change the accuracy of where that force is applied: doing automation thoughtlessly can create as many problems as it solves. Therefore, while we believe that software-based automation is superior to manual operation in most circumstances, better than either option is a higher-level system design requiring neither of them—an *autonomous* system. Or to put it another way, the value of automation comes from both what it does and its judicious application. We'll discuss both the value of automation and how our attitude has evolved over time.

The Value of Automation

What exactly is the value of automation?[1]

Consistency

Although scale is an obvious motivation for automation, there are many other reasons to use it. Take the example of university computing systems, where many systems engineering folks started their careers. Systems administrators of that background were generally charged with running a collection of machines or some

1 For readers who already feel they precisely understand the value of automation, skip ahead to "The Value for Google SRE" on page 70. However, note that our description contains some nuances that might be useful to keep in mind while reading the rest of the chapter.

software, and were accustomed to manually performing various actions in the discharge of that duty. One common example is creating user accounts; others include purely operational duties like making sure backups happen, managing server failover, and small data manipulations like changing the upstream DNS servers' *resolv.conf*, DNS server zone data, and similar activities. Ultimately, however, this prevalence of manual tasks is unsatisfactory for both the organizations and indeed the people maintaining systems in this way. For a start, any action performed by a human or humans hundreds of times won't be performed the same way each time: even with the best will in the world, very few of us will ever be as consistent as a machine. This inevitable lack of consistency leads to mistakes, oversights, issues with data quality, and, yes, reliability problems. In this domain—the execution of well-scoped, known procedures—the value of consistency is in many ways the primary value of automation.

A Platform

Automation doesn't just provide consistency. Designed and done properly, automatic systems also provide a *platform* that can be extended, applied to more systems, or perhaps even spun out for profit.[2] (The alternative, no automation, is neither cost effective nor extensible: it is instead a tax levied on the operation of a system.)

A platform also *centralizes mistakes*. In other words, a bug fixed in the code will be fixed there once and forever, unlike a sufficiently large set of humans performing the same procedure, as discussed previously. A platform can be extended to perform additional tasks more easily than humans can be instructed to perform them (or sometimes even realize that they have to be done). Depending on the nature of the task, it can run either continuously or much more frequently than humans could appropriately accomplish the task, or at times that are inconvenient for humans. Furthermore, a platform can export metrics about its performance, or otherwise allow you to discover details about your process you didn't know previously, because these details are more easily measurable within the context of a platform.

Faster Repairs

There's an additional benefit for systems where automation is used to resolve common faults in a system (a frequent situation for SRE-created automation). If automation runs regularly and successfully enough, the result is a reduced mean time to repair (MTTR) for those common faults. You can then spend your time on other tasks instead, thereby achieving increased developer velocity because you don't have to spend time either preventing a problem or (more commonly) cleaning up after it.

2 The expertise acquired in building such automation is also valuable in itself; engineers both deeply understand the existing processes they have automated and can later automate novel processes more quickly.

As is well understood in the industry, the later in the product lifecycle a problem is discovered, the more expensive it is to fix; see Chapter 17. Generally, problems that occur in actual production are most expensive to fix, both in terms of time and money, which means that an automated system looking for problems as soon as they arise has a good chance of lowering the total cost of the system, given that the system is sufficiently large.

Faster Action

In the infrastructural situations where SRE automation tends to be deployed, humans don't usually react as fast as machines. In most common cases, where, for example, failover or traffic switching can be well defined for a particular application, it makes no sense to effectively require a human to intermittently press a button called "Allow system to continue to run." (Yes, it is true that sometimes automatic procedures can end up making a bad situation worse, but that is why such procedures should be scoped over well-defined domains.) Google has a large amount of automation; in many cases, the services we support could not long survive without this automation because they crossed the threshold of manageable manual operation long ago.

Time Saving

Finally, time saving is an oft-quoted rationale for automation. Although people cite this rationale for automation more than the others, in many ways the benefit is often less immediately calculable. Engineers often waver over whether a particular piece of automation or code is worth writing, in terms of effort saved in not requiring a task to be performed manually versus the effort required to write it.[3] It's easy to overlook the fact that once you have encapsulated some task in automation, anyone can execute the task. Therefore, the time savings apply across anyone who would plausibly use the automation. Decoupling operator from operation is very powerful.

Joseph Bironas, an SRE who led Google's datacenter turnup efforts for a time, forcefully argued:

"If we are engineering processes and solutions that are not automatable, we continue having to staff humans to maintain the system. If we have to staff humans to do the work, we are feeding the machines with the blood, sweat, and tears of human beings. Think *The Matrix* with less special effects and more pissed off System Administrators."

3 See the following XKCD cartoon: *http://xkcd.com/1205/*.

The Value for Google SRE

All of these benefits and trade-offs apply to us just as much as anyone else, and Google does have a strong bias toward automation. Part of our preference for automation springs from our particular business challenges: the products and services we look after are planet-spanning in scale, and we don't typically have time to engage in the same kind of machine or service hand-holding common in other organizations.[4] For truly large services, the factors of consistency, quickness, and reliability dominate most conversations about the trade-offs of performing automation.

Another argument in favor of automation, particularly in the case of Google, is our complicated yet surprisingly uniform production environment, described in Chapter 2. While other organizations might have an important piece of equipment without a readily accessible API, software for which no source code is available, or another impediment to complete control over production operations, Google generally avoids such scenarios. We have built APIs for systems when no API was available from the vendor. Even though purchasing software for a particular task would have been much cheaper in the short term, we chose to write our own solutions, because doing so produced APIs with the potential for much greater long-term benefits. We spent a lot of time overcoming obstacles to automatic system management, and then resolutely developed that automatic system management itself. Given how Google manages its source code [Pot16], the availability of that code for more or less any system that SRE touches also means that our mission to "own the product in production" is much easier because we control the entirety of the stack.

Of course, although Google is ideologically bent upon using machines to manage machines where possible, reality requires some modification of our approach. It isn't appropriate to automate every component of every system, and not everyone has the ability or inclination to develop automation at a particular time. Some essential systems started out as quick prototypes, not designed to last or to interface with automation. The previous paragraphs state a maximalist view of our position, but one that we have been broadly successful at putting into action within the Google context. In general, we have chosen to create platforms where we could, or to *position* ourselves so that we could create platforms over time. We view this platform-based approach as necessary for manageability and scalability.

The Use Cases for Automation

In the industry, *automation* is the term generally used for writing code to solve a wide variety of problems, although the motivations for writing this code, and the solutions

4 See, for example, *http://blog.engineyard.com/2014/pets-vs-cattle*.

themselves, are often quite different. More broadly, in this view, automation is "meta-software"—software to act on software.

As we implied earlier, there are a number of use cases for automation. Here is a non-exhaustive list of examples:

- User account creation
- Cluster turnup and turndown for services
- Software or hardware installation preparation and decommissioning
- Rollouts of new software versions
- Runtime configuration changes
- A special case of runtime config changes: changes to your dependencies

This list could continue essentially *ad infinitum*.

Google SRE's Use Cases for Automation

In Google, we have all of the use cases just listed, and more.

However, within Google SRE, our primary affinity has typically been for running infrastructure, as opposed to managing the quality of the data that passes over that infrastructure. This line isn't totally clear—for example, we care deeply if half of a dataset vanishes after a push, and therefore we alert on coarse-grain differences like this, but it's rare for us to write the equivalent of changing the properties of some arbitrary subset of accounts on a system. Therefore, the context for our automation is often automation to manage the lifecycle of systems, not their data: for example, deployments of a service in a new cluster.

To this extent, SRE's automation efforts are not far off what many other people and organizations do, except that we use different tools to manage it and have a different focus (as we'll discuss).

Widely available tools like Puppet, Chef, cfengine, and even Perl, which all provide ways to automate particular tasks, differ mostly in terms of the level of abstraction of the components provided to help the act of automating. A full language like Perl provides POSIX-level affordances, which in theory provide an essentially unlimited scope of automation across the APIs accessible to the system,[5] whereas Chef and Puppet provide out-of-the-box abstractions with which services or other higher-level entities can be manipulated. The trade-off here is classic: higher-level abstractions are easier to manage and reason about, but when you encounter a "leaky abstraction,"

5 Of course, not every system that needs to be managed actually provides callable APIs for management—forcing some tooling to use, e.g., CLI invocations or automated website clicks.

you fail systemically, repeatedly, and potentially inconsistently. For example, we often assume that pushing a new binary to a cluster is atomic; the cluster will either end up with the old version, or the new version. However, real-world behavior is more complicated: that cluster's network can fail halfway through; machines can fail; communication to the cluster management layer can fail, leaving the system in an inconsistent state; depending on the situation, new binaries could be staged but not pushed, or pushed but not restarted, or restarted but not verifiable. Very few abstractions model these kinds of outcomes successfully, and most generally end up halting themselves and calling for intervention. Truly bad automation systems don't even do that.

SRE has a number of philosophies and products in the domain of automation, some of which look more like generic rollout tools without particularly detailed modeling of higher-level entities, and some of which look more like languages for describing service deployment (and so on) at a very abstract level. Work done in the latter tends to be more reusable and be more of a common platform than the former, but the complexity of our production environment sometimes means that the former approach is the most immediately tractable option.

A Hierarchy of Automation Classes

Although all of these automation steps are valuable, and indeed an automation platform is valuable in and of itself, in an ideal world, we wouldn't need externalized automation. In fact, instead of having a system that *has* to have external glue logic, it would be even better to have a system that needs *no glue logic at all*, not just because internalization is more efficient (although such efficiency is useful), but because it has been designed to not need glue logic in the first place. Accomplishing that involves taking the use cases for glue logic—generally "first order" manipulations of a system, such as adding accounts or performing system turnup—and finding a way to handle those use cases directly within the application.

As a more detailed example, most turnup automation at Google is problematic because it ends up being maintained separately from the core system and therefore suffers from "bit rot," i.e., not changing when the underlying systems change. Despite the best of intentions, attempting to more tightly couple the two (turnup automation and the core system) often fails due to unaligned priorities, as product developers will, not unreasonably, resist a test deployment requirement for every change. Secondly, automation that is crucial but only executed at infrequent intervals and therefore difficult to test is often particularly fragile because of the extended feedback cycle. Cluster failover is one classic example of infrequently executed automation: failovers might only occur every few months, or infrequently enough that inconsistencies between instances are introduced. The evolution of automation follows a path:

1) No automation
: Database master is failed over manually between locations.

2) Externally maintained system-specific automation
: An SRE has a failover script in his or her home directory.

3) Externally maintained generic automation
: The SRE adds database support to a "generic failover" script that everyone uses.

4) Internally maintained system-specific automation
: The database ships with its own failover script.

5) Systems that don't need any automation
: The database notices problems, and automatically fails over without human intervention.

SRE hates manual operations, so we obviously try to create systems that don't require them. However, sometimes manual operations are unavoidable.

There is additionally a subvariety of automation that applies changes not across the domain of specific system-related configuration, but across the domain of production as a whole. In a highly centralized proprietary production environment like Google's, there are a large number of changes that have a non–service-specific scope—e.g., changing upstream Chubby servers, a flag change to the Bigtable client library to make access more reliable, and so on—which nonetheless need to be safely managed and rolled back if necessary. Beyond a certain volume of changes, it is infeasible for production-wide changes to be accomplished manually, and at some time before that point, it's a waste to have manual oversight for a process where a large proportion of the changes are either trivial or accomplished successfully by basic relaunch-and-check strategies.

Let's use internal case studies to illustrate some of the preceding points in detail. The first case study is about how, due to some diligent, far-sighted work, we managed to achieve the self-professed nirvana of SRE: to automate ourselves out of a job.

Automate Yourself Out of a Job: Automate ALL the Things!

For a long while, the Ads products at Google stored their data in a MySQL database. Because Ads data obviously has high reliability requirements, an SRE team was charged with looking after that infrastructure. From 2005 to 2008, the Ads Database mostly ran in what we considered to be a mature and managed state. For example, we had automated away the worst, but not all, of the routine work for standard replica replacements. We believed the Ads Database was well managed and that we had harvested most of the low-hanging fruit in terms of optimization and scale. However, as daily operations became comfortable, team members began to look at the next level

of system development: migrating MySQL onto Google's cluster scheduling system, Borg.

We hoped this migration would provide two main benefits:

- *Completely* eliminate machine/replica maintenance: Borg would automatically handle the setup/restart of new and broken tasks.
- Enable bin-packing of multiple MySQL instances on the same physical machine: Borg would enable more efficient use of machine resources via Containers.

In late 2008, we successfully deployed a proof of concept MySQL instance on Borg. Unfortunately, this was accompanied by a significant new difficulty. A core operating characteristic of Borg is that its tasks move around automatically. Tasks commonly move within Borg as frequently as once or twice per week. This frequency was tolerable for our database replicas, but unacceptable for our masters.

At that time, the process for master failover took 30–90 minutes per instance. Simply because we ran on shared machines and were subject to reboots for kernel upgrades, in addition to the normal rate of machine failure, we had to expect a number of otherwise unrelated failovers every week. This factor, in combination with the number of shards on which our system was hosted, meant that:

- Manual failovers would consume a substantial amount of human hours and would give us best-case availability of 99% uptime, which fell short of the actual business requirements of the product.
- In order to meet our error budgets, each failover would have to take less than 30 seconds of downtime. There was no way to optimize a human-dependent procedure to make downtime shorter than 30 seconds.

Therefore, our only choice was to automate failover. Actually, we needed to automate more than just failover.

In 2009 Ads SRE completed our automated failover daemon, which we dubbed "Decider." Decider could complete MySQL failovers for both planned and unplanned failovers in less than 30 seconds 95% of the time. With the creation of Decider, MySQL on Borg (MoB) finally became a reality. We graduated from optimizing our infrastructure for a lack of failover to embracing the idea that failure is inevitable, and therefore optimizing to recover quickly through automation.

While automation let us achieve highly available MySQL in a world that forced up to two restarts per week, it did come with its own set of costs. All of our applications had to be changed to include significantly more failure-handling logic than before. Given that the norm in the MySQL development world is to assume that the MySQL instance will be the most stable component in the stack, this switch meant customizing software like JDBC to be more tolerant of our failure-prone environment. How-

ever, the benefits of migrating to MoB with Decider were well worth these costs. Once on MoB, the time our team spent on mundane operational tasks dropped by 95%. Our failovers were automated, so an outage of a single database task no longer paged a human.

The main upshot of this new automation was that we had a lot more free time to spend on improving other parts of the infrastructure. Such improvements had a cascading effect: the more time we saved, the more time we were able to spend on optimizing and automating other tedious work. Eventually, we were able to automate schema changes, causing the cost of total operational maintenance of the Ads Database to drop by nearly 95%. Some might say that we had successfully automated ourselves out of this job. The hardware side of our domain also saw improvement. Migrating to MoB freed up considerable resources because we could schedule multiple MySQL instances on the same machines, which improved utilization of our hardware. In total, we were able to free up about 60% of our hardware. Our team was now flush with hardware and engineering resources.

This example demonstrates the wisdom of going the extra mile to deliver a platform rather than replacing existing manual procedures. The next example comes from the cluster infrastructure group, and illustrates some of the more difficult trade-offs you might encounter on your way to automating *all* the things.