Google

How Google SREs Modify Production Resources Securely & Safely

DevOps Enterprise Summit 2022

Brett Beekley, **Michael Bird** - Site Reliability Engineers

sre.google • twitter.com/googlesre

Site Reliability Engineering

Our paths to Site Reliability Engineering

Brett

Academic Research: Learned how to rapidly create+test hypotheses.

Tech consulting: Learned how to listen to users and gather requirements.

Startup: Learned how to scale up systems 10-100x.

Google: SRE in Privacy & Security

Michael

After Undergrad: Compiler Engineer for Embedded Systems

Consulting Startup: Co-founder; Head of Technology

Payments Ecosystem: MTS Lead for adoption of Service Oriented Architecture

Google: Ads Infrastructure → YouTube Infrastructure → SRE in Privacy & Security

This presentation is rooted in our experiences and training, but doesn't necessary reflect every process at Google.

What does our team do?

We are:

- A group of SREs, software engineers, and staff.
- Responsible for some of the services that support Google's corporate and Cloud security.
    - Such as auth*, DDoS protection, and secrets management.

If our services are unavailable, at best Googlers and users can't access their stuff, and at worst it's a security incident.

What is Site Reliability Engineering (SRE)?

TL;DR: SRE is a framework that applies a software engineering approach to operating large scale systems reliably.

Why don't we automate everything?

1. Time is a finite resource.
2. We can't account for (and therefore automate) everything that might happen in production.

→ Therefore, humans must still be able to make production changes.

Problem: Human access to production opens reliability, privacy, and security risk if they make a mistake or their account is compromised.

How can we give SREs access to production while minimizing the risk of manual changes?

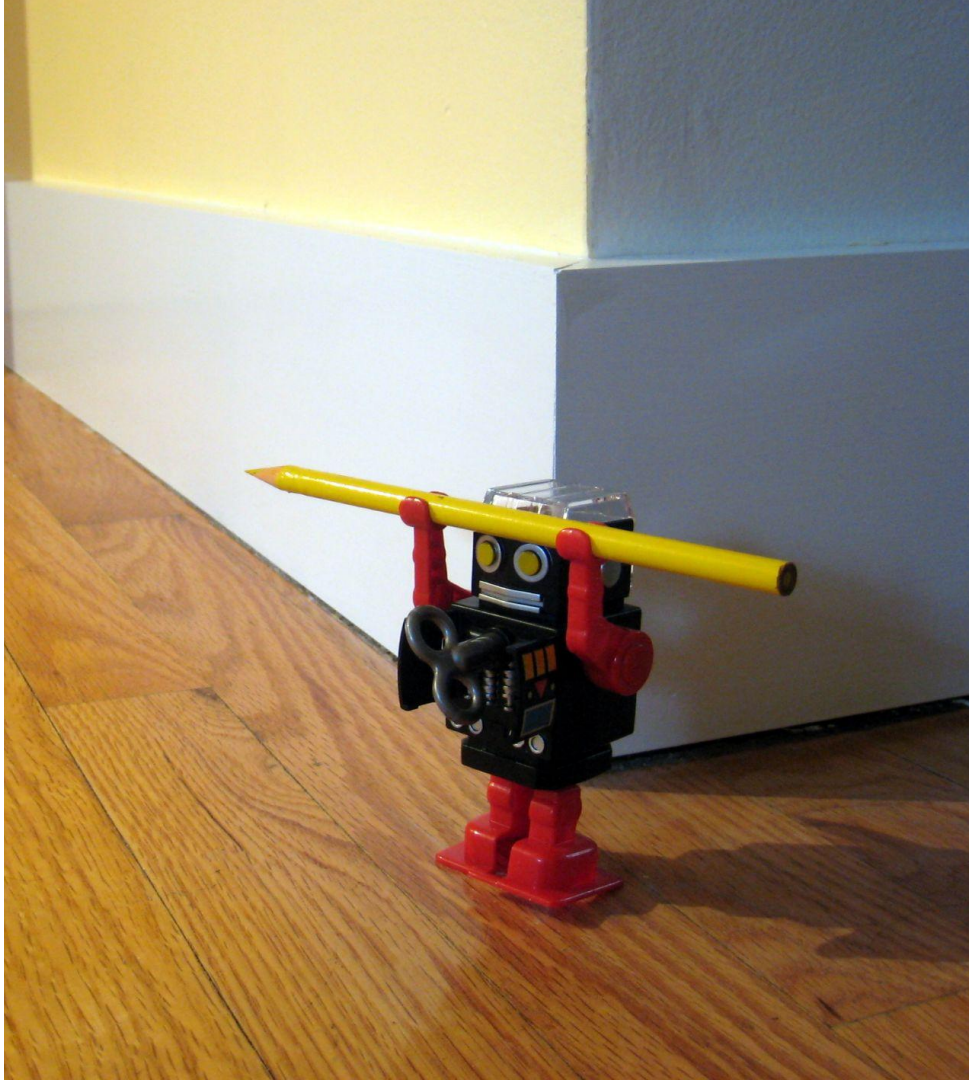
General solution

Establish a prioritized list of methods for changing production.

1. Automation
2. Tool-guided, manual intervention
3. Individual, peer-reviewed changes
4. Direct access

1. Automation

- Vast majority of production changes are automated.
- Automation requires validation that it works, which itself can be automated.



Case Study - Updating network rules for new machines

Problem: When updating hardware, some of our services require new network rules that allow traffic for the new machines' IP addresses.

Solution:

- Built a pipeline that ingests IP addresses of new machines and publishes the corresponding firewall rules.

Result: No human involvement necessary to add new machines to the network.

2. Tool-guided, manual intervention

- Humans perform a small minority of changes through on-rails tools.
- This is toil.
- If **automation doesn't yet support** the process, this is a "feature request"
- If **automation failed** or missed something, this is a "bug".

Case Study - Manual release testing

Problem: One user journey involves a physical button press as a trigger.

Solution:

- Automated everything except triggering the test itself
- Attached a response SLO to the toil of test execution, to keep the tester accountable but not rushed.
- Filed a feature request to build a virtual interface that can be programmatically "touched".

Result: A few minutes of toil per week has caught bugs before they impact users.

3. Individual, peer-reviewed changes

- Humans perform a tiny fraction of changes by getting a 2nd factor approval for individual commands.
- If **no tool exists yet**, this is a feature request.
- If **the tool failed**, this is an outage.
- Either way, a fix is high priority.

Case Study - Manual, but infrequent changes.

Problem: We rotate a set of credentials annually through a manual process.

Solution:

- Automated with off-the-shelf tooling where possible.
- More-complex cases would require a custom solution. Instead, keep it manual, but added a 2nd-factor approval process.

Result: Automation would save O(hours/year) of toil at best. It is still considered a "bug", but with low priority.

4. "Breaking glass" and bypassing all of the above

- Humans almost never directly, unilaterally modify production.
- This is defense-in-depth to make sure SRE is never locked out of production during a broad incident.

Where we are now

After enforcing this policy, our immediate team:

1. Decreased our manual production access by half.
2. Has ~zero people with "ambient" unilateral access to production.

Source: Analysis of internal data, 2021-2022.

Next Steps

Improve tool coverage: Move more production changes up the prioritized list toward automation.

Improve tool reliability: Decrease chance of humans needing to bypass automation, and moving down the prioritized list toward toil.

Summary

Type of Production Access	Frequency	Work to make possible
Automation	Vast majority	Build automated tools with monitoring to trigger and validate them.
Tool-guided, manual intervention	Small minority	Build control planes for those automated tools.
Individual, peer-reviewed changes	Tiny fraction	Build processes to review production changes
Break-glass direct access	Almost never	Build an emergency pathway to production and monitor its use.

Key Takeaway: We use a software engineering approach to decide how to prioritize work that iteratively approaches zero human access to production.

Thank you!

Emergency Incident Response

Planet-Scale Distributed Systems

Service Level Objectives (SLOs)

Systems Engineering

Global Storage

Load Balancing

Google

Monitoring

Availability

Embracing Risk

Blameless Failures

Site Reliability Engineering

Software Engineering

Automation

"Hope Is Not A Strategy"

sre.google