Have you tried turning it off ...and not on again?

Josh Deprez, Google Australia
@DrJosh9000

SREcon18 EMEA, 2018-08-31
About me

SRE at Google in Australia for 4 years, 8 months

PhD in mathematics

Retrocomputing enthusiast
What this talk is not about
The Great Wave Off Kanagawa, Hokusai, c1829-1833
Deprecation
Turn-down versus Deprecation

**Turn-down:**
- Sunsetting
- Restricting access
- Reducing capacity
- Decommissioning
- Actually turning things off

Can be difficult, and deserving recognition!

**Deprecation:**
- Telling people to stop using it
- It still exists
- It's still supported
- Working on a replacement (maybe)

Less so.
The Curse of the Deprecated System
Curse of the Deprecated System

Keep out!

Or enter.

I'm a presentation slide, not a cop.

(Apologies to Groening, et al.)
Curse of the Deprecated System

- Still supported and can't be turned off, it has users depending on it
- Alternative new system is usually "not ready yet"
- Teams see the existing legacy system as better supported
- New systems developed against the deprecated system, not the replacement
- The deprecated system grows more reverse dependencies
- Becoming harder to turn off...
Don't stop at deprecation
SRE and turn-down
Legacy systems are technical debt
Repaying technical debt improves reliability
Reliability is important to long-term velocity
Turning down legacy systems improves reliability and long-term velocity
Why SRE though?

SRE should have **empathy** for the service owners impacted by turn-downs, because turn-down is often related to outages in their own systems.

SRE usually has the **keys** to perform the turn-down, and uses supporting systems (automation, release procedures) for a contained implosion.
Service Reliability Hierarchy,
SRE and legacy systems

- If you have testing and release procedures, take advantage of them to perform the turn-down
  - Treat the turn-down like a release
- After turning down a system, write a postmortem
  - What went well, what went poorly, why we're never building this system again
- No alerting and monitoring = unreliable service = "no" service
  - Teams who care about your SLO should consider the service to be OFF at that moment.
  - But beware Hyrum's Law and the Curse of the Deprecated System
SRE and legacy systems

SRE should have the ability to reduce the toil / ops load (pages, etc)

Detuning monitoring, handing back to devs, etc are ways to do that.

Turning off legacy systems is also a legitimate way to do that!
Making turn-down easier

~ or ~

Fighting Hyrum's Law
Hyrum's Law

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."

(From www.hyrumslaw.com)
ACLs

*Before turning the service on* (or as soon as possible!), add an allow-list of users.

Ideally, enforce ACLs with something stronger than string-matching.

Record a contact email for each user in the list. This will help finding who to contact when time comes to turn it off.

Record context of each addition - ticket number / bug number
Contact emails

Good:
- A service-specific team list
- An onduty/interrupts rotation
- Leads list (managers + TLs)
- Oncall, if you must

Bad:
- An individual
  (Bus Number problem)
- A non-specific team list
  (Bystander Effect)
- A broad announcement list
  (high noise floor)
ACLs

Once a user migrates to the new service, +1 week (give or take): 
ACL them out of the legacy service!

Once no users remain in the ACL, good to turn down completely (right?)
Bazel & source code

"Visibility" - an allow-list for reverse dependencies (in terms of code)

Like an ACL for who can use your code.

(An aside:
Having the entire company's source code and configs in one searchable repository helps to find users.)
Good API design

Smaller interfaces are easier to reimplement or turn off: they limit the possible observable behaviours:

```go
define FetchKitten(name string) Kitten
func Authorize(user, resource string) bool

Flexible and broad interfaces are harder to turn off.

func KittensMatchingQuery(q string) []Kitten
func AllAuthzData() []Authorization
```
Logs

Resorting to logging requests is hardly ideal for finding users of a service.

If usage is low, log or observe everything about every request (modulo compliance requirements). User-Agent, source IP, ...

Analyse:

● Queries at regular intervals?
● Some things not queried?
Timeline of a turn-down
Timeline of a turndown
Timeline of a turndown

3 months

3 months

3 months
Timeline of a turndown

- Initial announcement
- Turn-down deadline announcement
- It's off!
- Working with affected users
- Implementing replacements and workarounds
- Minor fixes to workarounds or backout or extending deadline
- Monitoring for long tail of unmigrated users
Timeline of a turndown

The minimum lead time for infrastructure change mandates, such as turning off a system others is using, is **three months**.

Work must be scheduled, announcements must be repeated...
End of the line

Before the planned turndown date (1 month?), reduce capacity.

Should have fewer users already, and therefore be overprovisioned.

Consider rollback sequence:

1. Off
2. On with minimal capacity
3. On with full capacity

Faster to turn up minimal capacity (if you have to!)
End of the line

If supposedly all important users have migrated away, schedule a test which takes down the system, and see who complains.

Nobody? Great - leave it down!
End of the line

Just prior to finally switching it off...

- One final announcement email ("this is happening now!")
- Disable or silence the alerting
Communication
Communicating turn-down

Think like an advertiser...

● Repetition!
● Cross-platform
  ○ Email, Google Chat, memes, posters, billboards, flashing neon signs, skywriting...

At Google sometimes you have to repeat yourself until the message sinks in
On emails

Suggested email subject:

[ACTION REQUIRED] ${legacy_system} is being turned down on ${date}

Suggested second line of email:

"This is the first announcement of this change."
Turn-down monitoring
Monitor your turn-down

How do you monitor how many queries are not being served if the service is no longer serving them?

Answer: write a stub of the service

- Serve 410 Gone for endpoints that aren't reimplemented and won't ever be!
- Serve a redirect to the new service, if practical.

Add monitoring to the stub service

Roll out using existing release frameworks!
Wait a minute...

But if the stub serves redirects, isn't it another service to turn off?

Yes, but one you do not offer an SLO on:

- No alerting, no pages, no ops toil
- No damage if it gets turned off accidentally

Ensure you turn it off frequently and randomly (see again Hyrum's Law)!
Examples
Example: a cache server

Gets specific data from an upstream service, manipulates it, serves from RAM.

- Diverse range of client code using the cache
  - Some other services, some end-clients
  - Some clients pinned to *ancient* versions (built >5 years before, code long since deleted)
- **Data provided by the cache must continue to be available**
  - Some (dangerous!) functionality not reimplemented in new system
- Access to legacy system difficult to ACL down
  - Openly available (internally)
  - The replacement system did not make this mistake :-(
Example: a cache server

Naming things is hard.

Job names: acache.\texttt{frontend}, acache.\texttt{frontend2}

Job config files: \texttt{.../server.borg}, \texttt{.../frontend.borg}

DNS names: \texttt{acacheserver.eg.google.com}, \texttt{acacheserver2.eg.google.com}

Code paths: \texttt{.../acache/server/legacy}, \texttt{.../acache/server/frontend}
Example: a cache server

No choice but to announce widely and with a lead time of several months, since there were a broad range of users.

Delay to turn-down date, due to a large project which could not be rescheduled.

Finally set a mandate in March: service will turned off by end of June.
Example: a cache server

It's fundamentally a HTTP server...

Wrote a shim:

- "/query" -> redirects to new service
- "/bulk" -> 410 Gone

Monitoring (with no alerting) set up to watch long tail of forgotten clients
Example: a cache server
Example: authentication & authorisation system

- Used widely internally
- Paradoxically, no practical ACL mechanisms
- Painstaking identification of use-cases and announcements
- No like-for-like replacement built (or desirable), but alternatives were created

Down to just one reverse dependency (with no possible migration path).

Hmmm...
Example: authentication & authorisation system

Hand over the service to the one remaining reverse dependency!

Inter-team negotiations.
Example: fancy database

A database with a peculiar schema and an odd query language, but other systems existed that scaled better.

- Arbitrary queries made it hard to analyze use-cases
  - Some pathological cases
- Important users, but found workarounds for most large ones
- Contained sensitive data (e.g. password hashes)
- Originally replicated across hundreds of VMs for scale
  - Replication was flaky and continually broken
Example: fancy database

Multi-year ACL-based turn-down enforcement:

- Put users and programmatic access into a set of allow-lists
  - Row-level ACLs, schema-level ACLs, query type ACLs...
- Steadily removed access
- Removed parts of schema no longer used
- Could finally monitor remaining use-cases sensibly, and reduce capacity.

Now only runs a handful of replicas

Planning to replace with a more Cloudy solution Soon(TM)
Example: extra fancy DB, but with Ruby

The Ruby part is mostly a sidenote, but through lack of discipline it enabled all sorts of difficult-to-debug behaviour depended on by downstream systems.

The monolith:

- Inventory and asset management system, plus
- User and group system, plus
- Authorisation data, plus
- DNS management, plus
- Network management, plus...
Example: extra fancy DB, but with Ruby

Divide and conquer!

- Microservice re-architecture to identify disparate components.
- Pushing some reimplementation onto customer teams.

Owner team still doing lots of work in the replacement services.
Don't forget...
Don't forget to... do all the fun stuff

- Delete the documentation
- Delete the code
- Delete outdated TODOs in other code
- Delete the Puppet configs / Terraform / etc
- Delete the load balancer configs, DNS entries / CNAMEs / etc
- Delete the container image from the registry / **wipe the disks**
- Down the jobs, power off the servers
- Nuke it from orbit, just to be sure
- Mark the bugs as obsolete, close the component
- Write a postmortem
Summary
"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

-- Antoine de Saint-Exupéry
Thanks!
Stop, you've gone too far this time