Configuration is (riskier than?) Code

Jamie Wilkinson

(@jaqx0r)

A Site Reliability Engineer at Google

linux.conf.au Gold Coast 2020

The outline

1. Some observations about configuration change causing massive outages
   a. Examples from public outages in last few years
   b. There’s many anecdotes of people hating config, too
2. I think the Universal Turing Machine theorem applies to config, too
   a. What really is Config then?
   b. Config changes the beahviour of programs; this is like how interpreters work, on (usually) a less powerful language
3. What does the research show?
   a. Public research has weak results, small datasets, bad data, inconclusive
   b. My research internally on our postmortem database shows no strong evidence for config being riskier
4. Why isn’t it as high as we think?
   a. What practices does Google do that is mitigating this theoretical risk?

Observations

June 2016: Google Cloud Networking

https://status.cloud.google.com/incident/compute/16015

Google Compute Engine Incident #16015

Networking issue with Google Compute Engine services

Incident began at **2016-08-05 00:54** and ended at **2016-08-05 02:40** (all times are **US/Pacific**).

DATE	TIME	DESCRIPTION
Aug 09, 2016	07:21	SUMMARY:

On Friday 5 August 2016, some Google Cloud Platform customers experienced increased network latency and packet loss to Google Compute Engine (GCE), Cloud VPN, Cloud Router and Cloud SQL, for a duration of 99 minutes. If you were affected by

“...a new procedure for diverting traffic from the router was used. This procedure applied a new configuration that resulted in…”

Some Google Compute Engine TCP and UDP traffic had elevated latency. Most ICMP, ESP, AH and SCTP traffic inbound from outside the Google network was silently dropped, resulting in existing connections being dropped and new connections timing out on connect.

Most Google Cloud SQL first generation connections from sources external to Google failed with a connection timeout. Cloud SQL second generation connections may have seen higher latency but not failure.

Jan 2018: Google Compute Engine

https://status.cloud.google.com/incident/compute/18001

Google Compute Engine Incident #18001

The issue with Google Compute Engine has been resolved for all affected projects as of 20:30 US/Pacific.

Incident began at **2018-01-31 18:20** and ended at **2018-01-31 19:50** (all times are **US/Pacific**).

DATE	TIME	DESCRIPTION
Feb 07, 2018	10:05	On Wednesday 31 January 2017, some Google Cloud services experienced elevated errors and latency on operations that required inter-data center network traffic for a duration of 72 minutes. The impact was visible during three windows: between The root cause of this incident was an error in a configuration update to the system that allocates network capacity for traffic between Google data centers. To prevent a recurrence, we will improve the automated checks that we run on configuration changes to detect problems before release. We will be improving the monitoring of the canary to detect problems before global rollout of changes to the configuration.

“...an error in a configuration update to the system that allocates network capacity”

May 2018: Google BigQuery

https://status.cloud.google.com/incident/bigquery/18036

Google BigQuery Incident #18036

Multiple failing BigQuery job types

Incident began at **2018-05-16 16:00** and ended at **2018-05-16 18:18** (all times are **US/Pacific**).

“Configuration changes being rolled out on the evening of the incident were not applied in the intended order. This resulted in an incomplete configuration change becoming live in some zones…

During the rollback attempt, another bad configuration change was enqueued for automatic rollout and when unblocked, proceeded to roll out…”

On Wednes… and query jobs for a duration of 88 minutes over … which was isolated to the EU). We sincerely apologize to all of our affected customers; this is not the level of reliability we aim to provide in our products. We will be issuing

On Wednesday 16 May 2018 from 16:00 to 16:55 and from to 17:45 to 18:18 PDT, Google BigQuery experienced a failure of some import, export and query jobs. During the first period of impact, there was a 15.26% job failure rate; during the second, which was isolated to the EU, there was a 2.23% error rate. Affected jobs would have failed with INTERNAL_ERROR as the reason.

Aug 2018: Google Cloud Networking

https://status.cloud.google.com/incident/cloud-networking/18013

Google Cloud Networking Incident #18013

We are investigating issues with Internet access for VMs in the europe-west4 region.

Incident began at **2018-07-27 18:27** and ended at **2018-07-27 19:31** (all times are **US/Pacific**).

DATE	TIME	DESCRIPTION
Aug 07, 2018	14:51	ISSUE SUMMARY

On Friday 27 July 2018, for a duration of 1 hour 4 minutes, Google Compute Engine (GCE) instances and Cloud VPN tunnels in europe-west4 experienced loss of connectivity to the Internet. The incident affected all new or recently live migrated GCE instances. VPN tunnels created during the incident were also impacted. We apologize to our customers whose services or

"...caused by an unintended side effect of a configuration change made to jobs that are critical in coordinating the availability..."

DETAILED DESCRIPTION OF IMPACT

19:31 PDT lost connectivity to the Internet and other instances via their public IP addresses. Additionally any instances that live migrated during the outage period would have lost connectivity for approximately 30 minutes after the live migration completed. All Cloud VPN tunnels created during the impact period, and less than 1% of existing tunnels in europe-west4 also lost external connectivity. All other instances and VPN tunnels continued to serve traffic. Inter-instance traffic via private IP addresses remained unaffected.

ROOT CAUSE

Google's datacenters utilize software load balancers known as Maglevs [1] to efficiently load balance network traffic [2] across

Nov 2018: Amazon EC2

https://aws.amazon.com/message/74876/

Summary of the Amazon EC2 DNS Resolution Issues in the Asia Pacific (Seoul) Region (AP-NORTHEAST-2)

한국어로 읽기

We'd like to give you some additional information about the service disruption that occurred in the Seoul (AP-NORTHEAST-2) Region on November 22, 2018. Between 8:19 AM and 9:43 AM KST, EC2 instances experienced DNS resolution issues in the AP-NORTHEAST-2 region. This was caused by a reduction in the number of healthy hosts that were part of the EC2 DNS resolver fleet, which provides a recursive DNS service to EC2 instances. Service was restored when the number of healthy hosts was restored to previous levels. EC2 network connectivity and DNS resolution outsi

"The root cause of DNS issues was a configuration update…"

The root cause of DNS resolution issues was a configuration update which incorrectly removed the setting that specifies the minimum healthy hosts for the EC2 DNS resolver fleet in the AP-NORTHEAST-2 Region. This resulted in the minimum healthy hosts configuration setting being interpreted as a very low default value that resulted in fewer in-service healthy hosts. With the reduced healthy host capacity for the EC2 DNS resolver fleet, DNS queries from within EC2 instances began to fail. At 8:21 AM KST, the engineering team was alerted to the DNS resolution issue within the AP-NORTHEAST-2 Region and immediately began working on resolution. We identified root cause at 8:48 AM KST and we first ensured that there was no further impact by preventing additional healthy hosts from being removed from service; this took an additional 15 minutes. We then started restoring capacity to previous levels which took the bulk of the recovery time. At 9:43 AM KST, DNS queries from within EC2 instances saw full recovery.

March 2019: Google Cloud Storage

https://status.cloud.google.com/incident/storage/19002

Google Cloud Storage Incident #19002

Elevated error rate with Google Cloud Storage.

Incident began at **2019-03-12 18:40** and ended at **2019-03-12 22:50** (all times are **US/Pacific**).

DATE	TIME	DESCRIPTION

"...a configuration change which had a side effect of overloading a key part of the system."

and 10 minutes. We apologize to custo… or application was impacted by this incident. We know that our customers depend on Google Cloud Pl… we are taking immediate steps to improve our availability and prevent outages of this type from recurring.

DETAILED DESCRIPTION OF IMPACT

On Tuesday 12 March 2019 from 18:40 to 22:50 PDT, Google's internal blob (large data object) storage service experienced elevated error rates, averaging 20% error rates with a short peak of 31% errors during the incident. User-visible Google services including Gmail, Photos, and Google Drive, which make use of the blob storage service also saw elevated error rates, although (as was the case with GCS) the user impact was greatly reduced by caching and redundancy built into those services. There will be a separate incident report for non-GCP services affected by this incident.

June 2019: Google Cloud

https://cloud.google.com/blog/topics/inside-google-cloud/an-update-on-sundays-service-disruption

INSIDE GOOGLE CLOUD

An update on Sunday's service disruption

Benjamin Treynor Sloss
VP, 24x7

June 4, 2019

Yesterday, a disruption in Google's network in parts of the United States caused slow performance and elevated error rates on several Google services, including Google Cloud Platform, YouTube, Gmail, Google Drive and others. Because the disruption reduced regional network capacity, the worldwide user impact varied widely. For most Google users there was little or no visible change to their services—search queries might have been a fraction of a second slower than usual for a few minutes but soon returned to normal, their Gmail continued to operate without a hiccup, and so on. However, for users

"In essence, the root cause of Sunday's disruption was a configuration change that was intended for a small number of servers in the region."

make Google's services available to everyone around the world, and when we fall short of that goal—as we did yesterday—we take it very seriously. The rest of this document explains briefly what happened, and what we're going to do about it.

Incident, Detection and Response

In essence, the root cause of Sunday's disruption was a configuration change that was intended for a small number of servers in a single region. The configuration was incorrectly applied to a larger number of servers across several neighboring regions, and it caused those regions to stop using more than half of their available network capacity

July 2019: CloudFlare

https://blog.cloudflare.com/cloudflare-outage/

Cloudflare outage caused by bad software deploy (updated)

"The cause of the outage was deployment of a single misconfigured rule within the Cloudflare Web Application Firewall (WAF) during a routine deployment..."

3 July 2019, 01:50 am

~~This is a short placeholder blog and will be replaced with a full post-mortem and disclosure of what happened today.~~

For about 30 minutes today, visitors to Cloudflare sites received 502 errors caused by a massive spike in CPU utilization on our network. This CPU spike was caused by a bad software deploy that was rolled back. Once rolled back the service returned to normal operation and all domains using Cloudflare returned to normal traffic levels.

The vox populi

Kelly Sommers
@kellabyte

Follow

“The massive outage was a result of a server configuration change” seems to be 90% of the massive outage descriptions. There’s either a lot to learn from that lesson or people are hiding a lot under “server configuration change”

1:26 PM - 14 Mar 2019

98 Retweets **401** Likes

https://twitter.com/kellabyte/status/1106290558618619904

Jez Humble @jezhumble

Follow

I mean, you're much more likely to catastrophically out your prod system with a config change than with a code change, so if you're not already checking them into version control and running them through a deployment pipeline, put down whatever you're doing right now...

> **Kelly Sommers** @kellabyte
> "The massive outage was a result of a server configuration change" seems to be 90% of the massive outage descriptions. There's either a lot to learn from that lesson or people are hiding a lot under "server configuration change"

1:43 PM - 14 Mar 2019

304 Retweets **619** Likes

https://twitter.com/jezhumble/status/1106294766587641856

Kent Beck
@KentBeck

Follow

Replying to @jezhumble

A ways into my time at Facebook I realized that the whole thing was programmed in config files and feature flag settings and everyone else was writing a gigantic interpreter for this strange language.

1:54 PM - 14 Mar 2019

72 Retweets **282** Likes

https://twitter.com/KentBeck/status/1106297538104061952

Matt Weagle
@mweagle

Follow

Related: “It’s just a config change.”

> **Deepak Singh** @mndoci
> OH: “it’s a trivial change”. Famous last words

11:59 AM - 20 May 2019

2 Retweets **17** Likes

https://twitter.com/mweagle/status/1130548712596430848

Configuration {as,is} code

(more thoughtleadership)

Rich Burroughs
@richburroughs

Follow

Just heard the phrase "coding in YAML" used unironically.

12:39 AM - 22 Dec 2018

2 Retweets **24** Likes

https://twitter.com/richburroughs/status/1076396782190391297

https://twitter.com/jaqx0r/status/1076402245489446913

Jamie Wilkinson
@jaqx0r

Follow

Replying to @richburroughs

Does the yaml tell a computer to do things? :)

1:36 AM - 22 Dec 2018

2 Likes

https://twitter.com/jaqx0r/status/1076411213813149696

What is configuration?

- Input parameters extracted from the program
- Distributed separately, faster, and more frequently than code
- A useful abstraction that hides details

Where does config come from?

- Command line options
- Files representing a data structure
- Environment variables
- User input: fields in the API request
- Administrative APIs, commands, and schemas

Example: Environment settings

```
period = os.getenv(“TRACE_PERIOD”, 0)

if period > 0:

  trace.SetTracePeriod(period)

if FLAGS.database == “”:

  log.Fatal(“no --database”)

InitDB(FLAGS_database)
```

```
$ export TRACE_PERIOD=1000

$ ./prog --database=”user:pass@dbhost”
```

Example: Feature flags

```
$ ./prog --enable_feature
```

```
if (FLAG_enable_feature) {
     CallSomeFeatureMethod();
}
```

```
$ ./prog --new_backend
```

```
if (FLAG_new_backend) {
  NewSchemaOrMicroservice();
} else {
  DoItTheOldWay();
}
```

Example: Thresholds

```
$ ./prog --max_connections=1000

…

func handleConnection(...) {

 if connCount > FLAG_max_connections {

   return 429;

 }

...

}
```

Inputs to a program

Given a program P

Let us describe our program with the function *P(x) -> y*

For some user input x it generates y. This is why you made it in the first place.

and a configuration C

We want the output of the program to be changed based on some administratively controlled input. This new input is called the configuration:

$$P(C, x) \rightarrow y$$

For a given configuration C, our program still turns an input x into an output y.

and a configuration C

We want the output of the program to be changed based on some administratively controlled input. This new input is called the configuration:

$$P(C, x) \rightarrow y$$

For a given configuration C, our program still turns an input x into an output y.

Some users might not be able to change C, they see a program Q:

$$Q(x) \rightarrow P(C, x) \rightarrow y$$

We can change the behaviour

without changing the program

Given a different configuration C’

$$P(C', x) \rightarrow y'$$

our output is now *y’*

This is not just theoretical

```
$ ./prog --enable_feature

if (FLAG_enable_feature) {

     FeatureMethod();

}
```

C = < *enable_feature: true* >

P(C, x) → *y* implies `FeatureMethod()`

C' = < *enable_feature: false* >

P(C', x) → *y'* implies **no** `FeatureMethod()`

C can represent a complex structure

C = <

enable_feature ∈ { true, false}

trace_period ∈ ℕ

address ∈ [A-Z]*

>

C = <O, E, F, S>

where

O = < *command line flags and options* >,

E = < *environment variables* >,

F = < *files* >,

S = < *state and schemas* >

We can be even more pedantic, too

$$P(C\langle O, E, F, S\rangle, I\langle A, x\rangle) \rightarrow y$$

where *P* and *C* are our previously defined function and configuration,

x is now joined with

A = user-provided API request configuration

in *I*, the whole user input to the program.

What else looks like this?

```
while !terminate {
  instr := fetch(pc)
   switch instr {
    case nop:
      break
    case jmp:
        pc = operand()
...
```

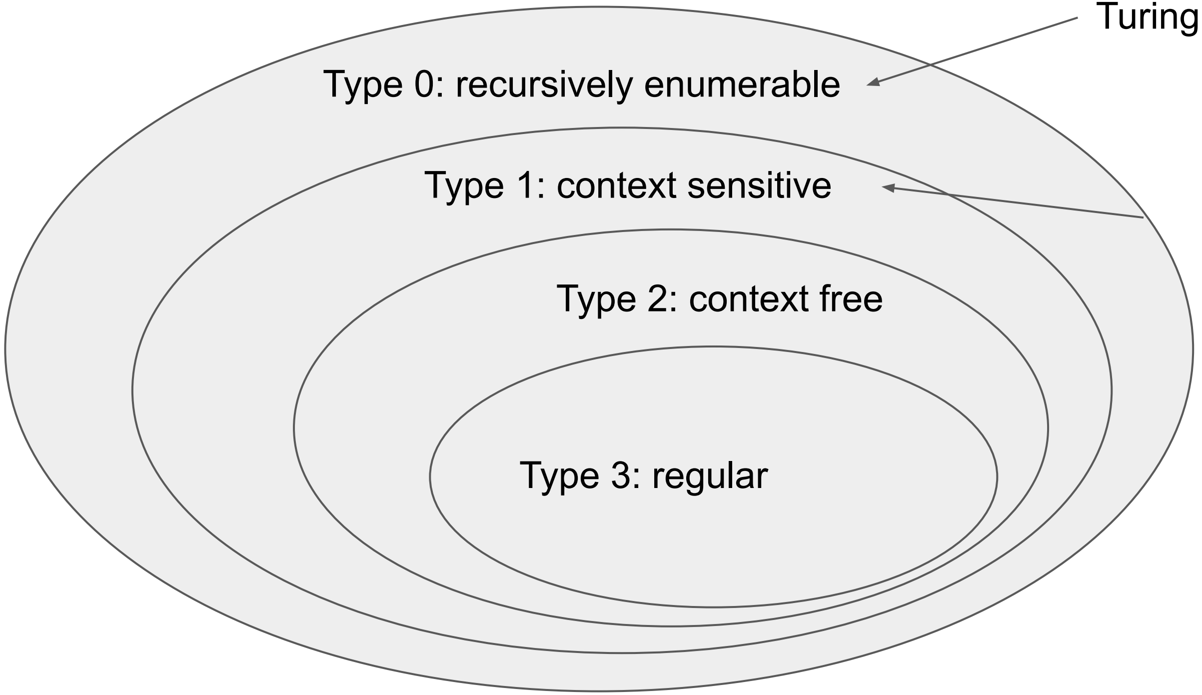
Simulators, virtual machines, interpreters take a configuration input (the program) and another input (the input to the program), and emit an output that changes with both the config and the input.

This is the Universal Turing Machine theorem

$P(C, x_1, ..., x_n) \rightarrow y$

$\Phi(\#P, C, x_1, \ldots, x_n) \rightarrow y$

“That’s not what Post-Turing says…”



We can move to strictly less powerful languages, but not back again.

With the right program, you can code in YAML

```
op: add
  left: 3
  right:
    op: add
      left: 1
      right: 2
```

The config language may *not be* Turing-complete, thus the program is not Turing-equivalent: that means you can't perform ***any*** computation*.

… but you *can* perform arbitrary computation within the range of the function *P*

*unless you're in infosec

Alternatively

Less powerful languages are still very useful, because their reduction in strength means we have more properties we can rely on

Your program is an interpreter.

(For a, perhaps, not very general language)

Configuration is code

(Not very powerful code,
but code nonetheless)

scawwy ewwiza
@mycoliza

Follow

who called it "infrastructure as code" and not "software-defined software"

8:27 AM - 10 Apr 2019 from Oakland, CA

7 Retweets **32** Likes

https://twitter.com/mycoliza/status/1115999705190649856

Brandon Hays
@tehviking

Follow

what idiot called it "YAML Parser Error" and not "A Series of Unfortunate Indents"

6:05 PM - 16 Apr 2018 from Austin, TX

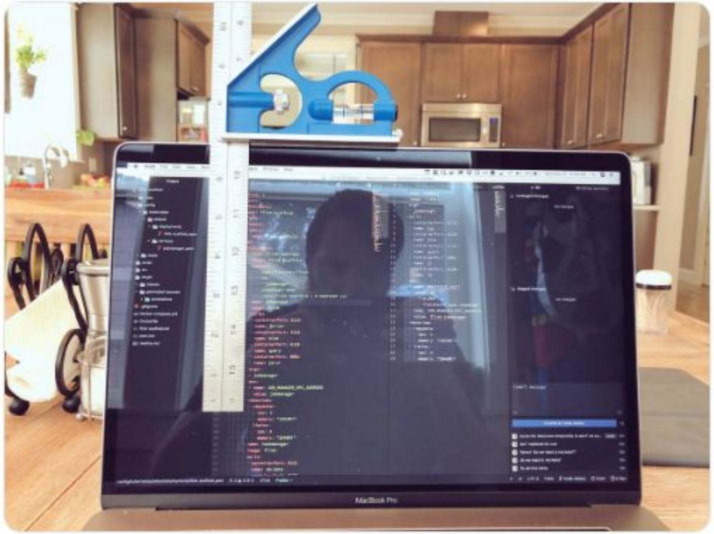
1,081 Retweets **3,799** Likes

https://twitter.com/tehviking/status/986048060487806976

Justin Palmer
@Caged

Follow

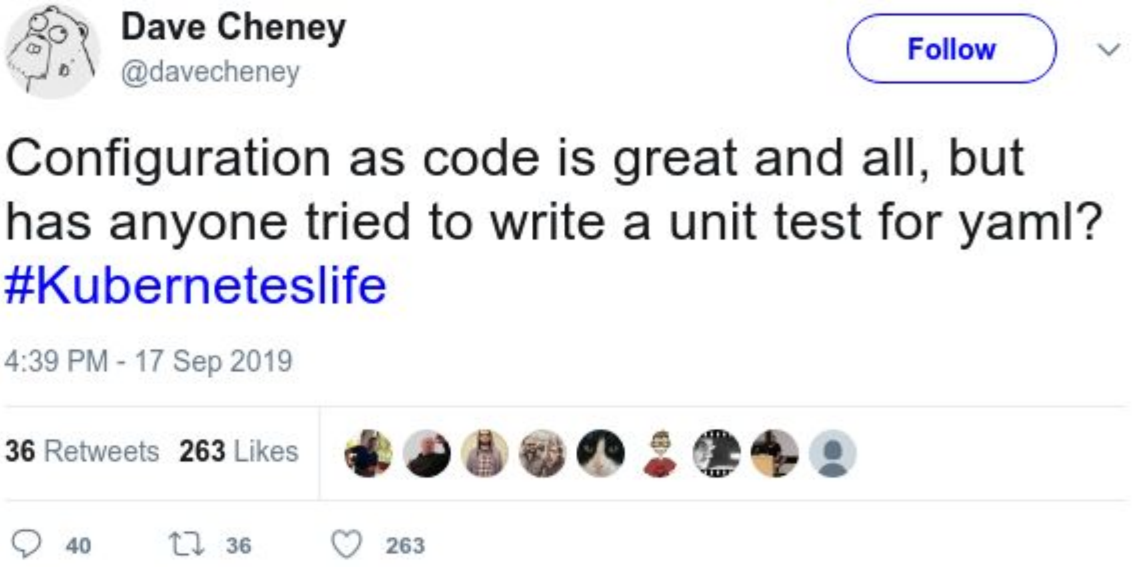
Writing YAML



11:01 AM - 12 Sep 2018

3,526 Retweets **9,916** Likes

https://twitter.com/Caged/status/1039937162769096704/photo/1


Dave Cheney
@davecheney
Follow

Configuration as code is great and all, but has anyone tried to write a unit test for yaml? #Kuberneteslife

4:39 PM - 17 Sep 2019

36 Retweets 263 Likes

40 36 263

https://twitter.com/davecheney/status/1174105604182269952

Caitie McCaffrey
@caitie

Follow

Replying to @kellabyte

I think one of the big challenges with config changes is they are difficult to test. Usually config changes are environment specific, so even if you have a CI/CD pipeline the prod config change only really gets tested in prod.

4:29 PM - 14 Mar 2019

16 Retweets **103** Likes

https://twitter.com/caitie/status/1106336594594680833

Jessica Joy Kerr
@jessitron

Follow

Replying to @PaulDJohnston @springrod and 3 others

It’s programming, except you can’t test it nearly as easy, or even check its syntax.
Pretty scary.
Config should be data, different from on env to env. Other than that let’s use code that is modular and testable please
That isn’t the world we live in ... yet

6:31 AM - 10 Apr 2019

6 Likes

https://twitter.com/jessitron/status/1115970585631371264

Reid D. M.
@arrdem

Follow

co-workers, sobbing: you can't just point at anything with a syntax tree and call it a lisp
me, pointing at a mountain of yaml: really shitty lisp

1:01 PM - 28 Oct 2019

23 Retweets **153** Likes

4 23 153

https://twitter.com/arrdem/status/1188908619435139072

James "pentagrames scarernbull" Turnbull @kartar · Jan 4

If only we had knowledge about some kind of Domain Specific Languages for configuration that would help us replace YAML and JSON...

22 8 43

Nathan L Smith @nlsmith · Jan 4

s-expressions

1 2

Adam Jacob
@adamhjk

Follow

Replying to @nlsmith @kartar

Dude, not even kidding. All this shit should've been Lisp.

12:41 PM - 4 Jan 2019

10 Likes

https://twitter.com/adamhjk/status/1081289520903942144

(This is what the GNU Scheme people have been saying for decades)

“Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

— *Philip Greenspun's tenth rule of programming*

Config programs evaluate to return parameters

```
((enable_feature t)

 (trace_period (cond (eq env “prod”)
1000 1))

 (address (concat “user:pass”

                (cond (eq env “prod”)

                   “dbhost”

                   “testdbhost”))
```

→

```
config <

   enable_feature: true

   trace_period: 1000

   address: “user:pass@dbhost”

>
```

(Should it have been LISP?)

Within Google there are:

- 110 “named” languages (including no-longer-used languages)
- 76 of these are “ordinary” (unspecialised), including JSON and Pythonic derivatives
  - Python is particularly juicy as a tool for expressing DSLs that trick you into thinking they’re Python. Rubyists might relate to this.
- Additionally there are
  - 72 more Yacc grammars
  - 466 ANTLR grammars
  - 92 lex programs, and
  - ~6000 occurrences of EBNF specifications
- None of these count command line flags that accept structured values
  - (e.g. text format protobufs)

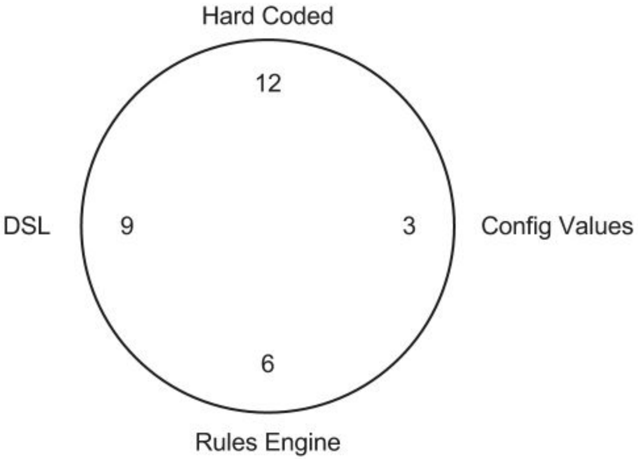
The Configuration Complexity Clock

http://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html

MONDAY, MAY 07, 2012

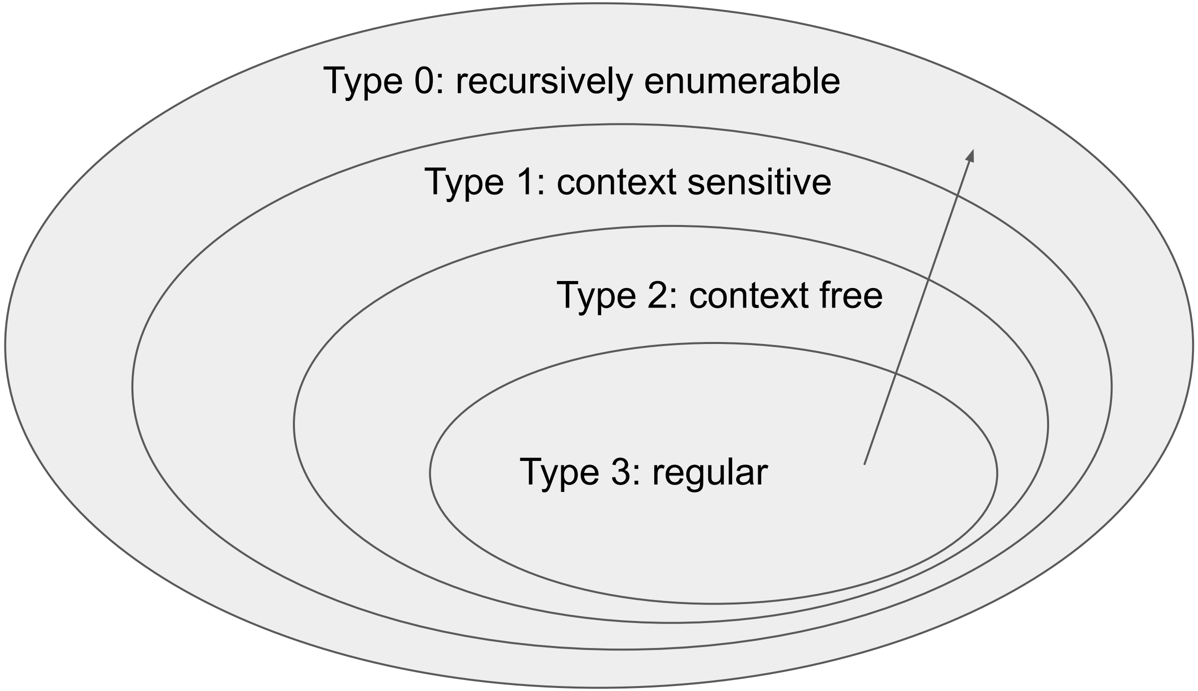
When I was a young coder, just starting out in the big scary world of enterprise software, an older, far more experienced chap gave me a stern warning about hard coding values in
some point, and you don't want to recompile and redeploy your application just to change the VAT tax rate." I took this advice to heart and soon every value that my application ne
still think it's good advice, but be warned, like most things in software, it's good advice *up to a point*. Beyond that point lies pain.

Let me introduce you to my 'Configuration Complexity Clock'.



This clock tells a story. We start at midnight, 12 o'clock, with a simple new requirement which we quickly code up as a little application. It's not expected to last very long, just a sto
we've hard-coded all the application's values. Months pass, the application becomes widely used, but there's a problem, some of the business values change, so we find ourselves re
change a few numbers. This is obviously wrong. The solution is simple, we'll move those values out into a configuration file, maybe some appsettings in our App.config. Now we're

Clock progression is increasing language power



We can move to more powerful langauges by creating constructs to express ourselves better.

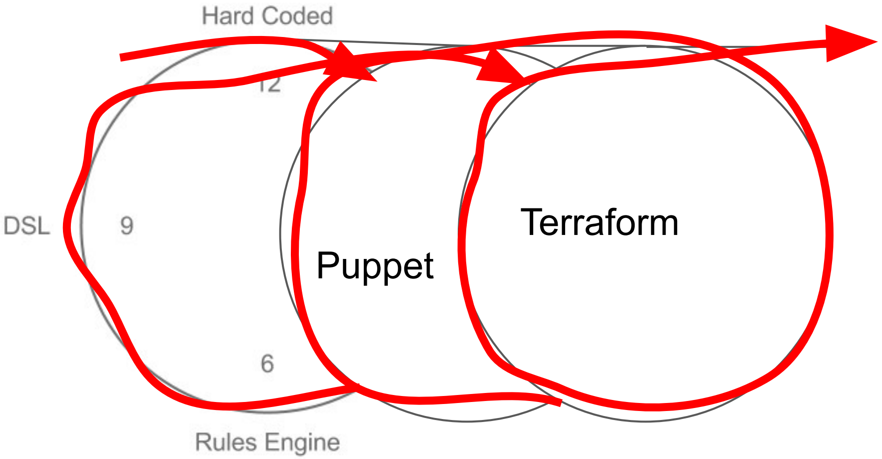
The Configuration Complexity Spiral

MONDAY, MAY 07, 2012

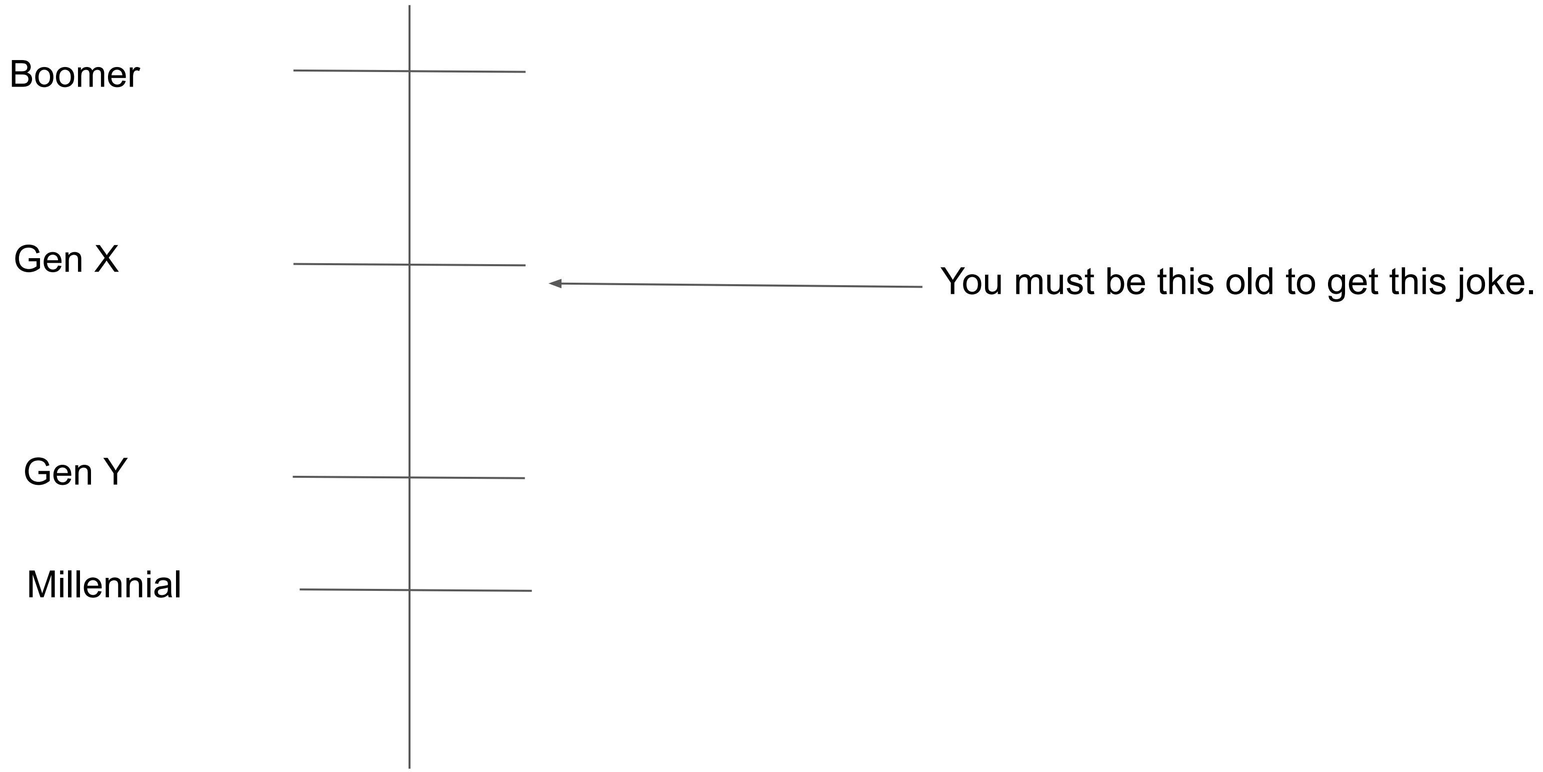
The Configuration Complexity Clock

When I was a young coder, just starting out in the big scary world of enterprise software, an older, far more experienced chap gave me a stern warning about hard coding values in
some point, and you don't want to recompile and redeploy your application just to change the VAT tax rate." I took this advice to heart and soon every value that my application ne
still think it's good advice, but be warned, like most things in software, it's good advice *up to a point*. Beyond that point lies pain.

Let me introduce you to my 'Configuration Complexity Clock'.



This clock tells a story. We start at midnight, 12 o'clock, with a simple new requirement which we quickly code up as a little application. It's not expected to last very long, just a sto
we've hard-coded all the application's values. Months pass, the application becomes widely used, but there's a problem, some of the business values change, so we find ourselves re
change a few numbers. This is obviously wrong. The solution is simple, we'll move those values out into a configuration file, maybe some appsettings in our App.config. Now we're



Oh sendmail.cf is also Turing complete.

Observation

$$P(C, x) \rightarrow y$$

Every configurable program has two users: the end user, and the administrator

“… but you *can* perform arbitrary computation within the range of the function *P*”

What’s the domain of *P*?

How many configuration options do you have?

$$P(C, x) \rightarrow y$$

The number of options in *C*:

$$|C|$$

How many values can they each take?

$$\prod_{i=1}^{n} |C_n|$$

https://pixabay.com/photos/shuttle-cockpit-space-rocket-642404/

A thesis:

Configuration:

- is like code
- is harder to test before production, because environment
- has larger force multipliers, thus larger impact per character, because of abstractions and automation
- is empirically the “cause” of several large publicly visible Cloud Outages

therefore Configuration:

1. will be a key factor in a majority of change related outages, and
2. as a key factor will correlate with higher severity outages

previous work

https://davidmytton.blog/what-are-the-common-causes-of-cloud-outages/ Not very conclusive; slight favour for config

https://people.cs.uchicago.edu/~shanlu/paper/hotos19_azure.pdf Uses different terminology, software bug causes, so the opposite side

Trends from Trenches: doesn’t break down cause by kind

SRE Book: 70% cause by change, not broken down by kind

Why does the cloud stop computing: problematic

Why Does the Cloud Stop Computing?

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA

597 public outages from 2009 to 2015

"Config" ranked 5th, 10% of "root causes". 3rd when limiting to "change"-like causes only.

Only classified an outage with a cause if the text contained the correct words.
Only classified each outage with a singular root cause.

What bugs cause production cloud incidents?

HotOS ’19, May 13–15, 2019, Bertinoro, Italy

Microsoft Azure based study.

Entirely different language for classifying cause.

That’s because it’s focussing on software defects, not change events.

The SRE Book

O’Reilly Media, 2016.

“SRE has found that roughly 70% of outages are due to changes in a live system.”

… and that’s it.

Incidents - Trends from the Trenches

https://m.subbu.org/incidents-trends-from-the-trenches-e2f8497d52ed

Feb 2019

Classifies based on “trigger”, the event that surfaced the outage.

A “large number” of outages covered.

Change is identified as a trigger in 1/3rd outages; and “software deployments” half of that.

“Config drift” is identified as trigger in 1/5th of outages, in which changes should have been applied to config, but have not.

What are the common causes of Cloud Outages?

https://davidmytton.blog/what-are-the-common-causes-of-cloud-outages/,

Jul 2019

49 public outage reports from 2016 to 2019.

16 attributed to “misconfiguration” (32%), 21 to bugs (43%)

4 to “human error”

A List Of Postmortems!

https://github.com/danluu/post-mortems#config-errors

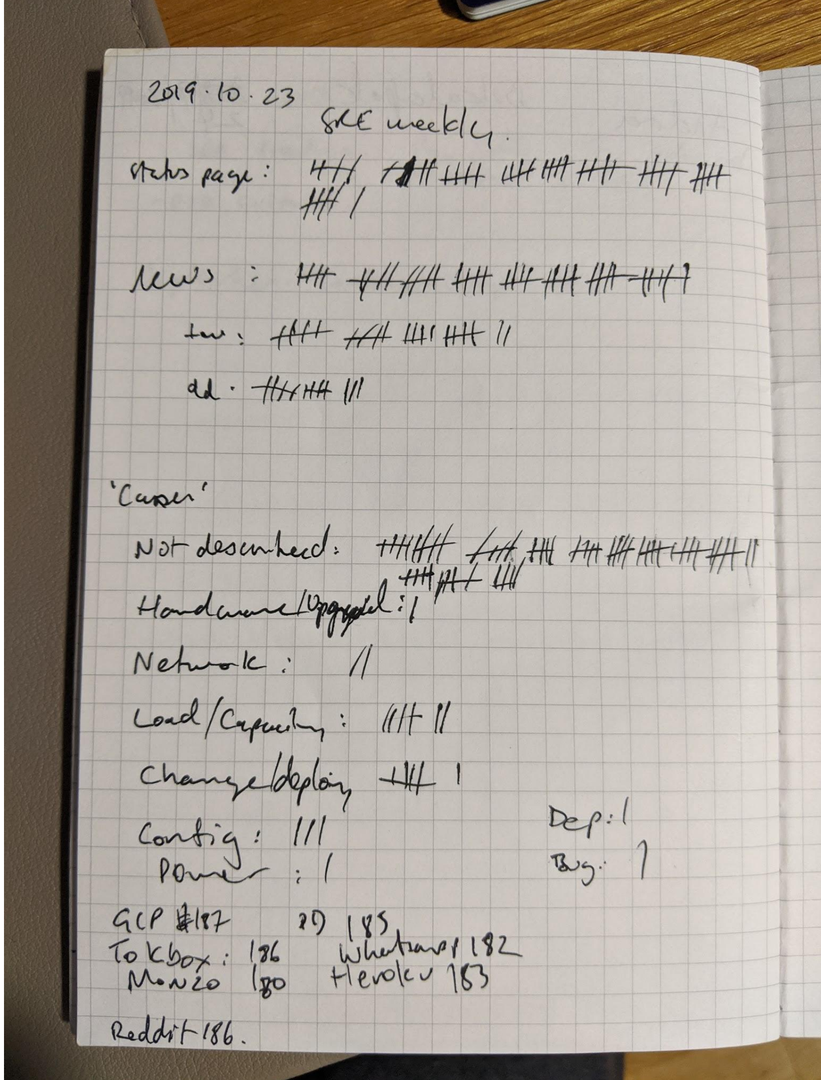
Community maintained list of postmortems, ~100 listed.

Configuration (21) ranked second after Uncategorized, no mention of software bugs.

My own research

Manually count SRE Weekly Newsletter from https://sreweekly.com/

Got bored, terrible data; mostly noise, about 1% of articles had useful information in it.



My own research, cont

Explore the Google postmortems dataset. Many thousands of reports of all severities dating back many years.

Multiple-choice classification of causes and triggers by author at creation time. Can manually keyword match against data.

Measured config push, binary push, both, and neither.

Config and Binary are equal in size; config is slightly higher than binary (by 2%) in when comparing only “big” severity outages.

Year over year, config was slightly higher up until 2018 when the pattern reversed, and in 2019 equal.

Results

Insufficient data from public studies to draw a strong conclusion.

Sufficient data from internal study to conclude that, internally, config and code are equally risky. This is actually somewhat reassuring because it is not in conflict with the theory that config is code.

Unsatisfying, possibly insignificant result that config is slightly more likely to be a cause in large outages than code. But looks like it was higher in the past.

So the theory is incorrect, or is Google an outlier and manages that risk well?

Risk Mitigations

There’s always low hanging fruit

Simple Testing Can Prevent Most Critical Failures (OSDI 2014) shows that simple testing can eliminate 1/5th outages in systems observed, lesson is there’s always low hanging fruit.

If config is code, and config changes are equally likely to cause an outage as code changes are, then config testing should be part of the CI/CD.

1. Simple parse test
2. Validation test (using same code as main program)

Config that is a program can perform assertions; all those less powerful languages need you to write the test program.

Put everything in version control

Everything, even “running a command against an API endpoint” (e.g. schema changes).

Make a script if necessary. Try to avoid “human runs a maintenance command from their workstation.”

Code review and audit logs address time to resolve incidents by having information about change more visible.

Recall the two users of any system

- Help the administrators make good decisions.
- The **sooner** a config is validated after **commit**, the better
  - Validation that happens only during deploy is better than nothing, but slow feedback loops lead to unhappy people
  - Factor out validation into small binaries to run during code review
- Configs that are the result of generators can show diffs against the last version in code review
  - Showing the author and reviewer the closest thing to “how the bare metal will change” improves understanding
  - Corollary: Config generators need investment in error reporting to aide the humans, rather than confuse them
- Automatic config formatting just like code formatters
  - Removes cognitive burdens when reviewing change

Staging/Pre-production environments

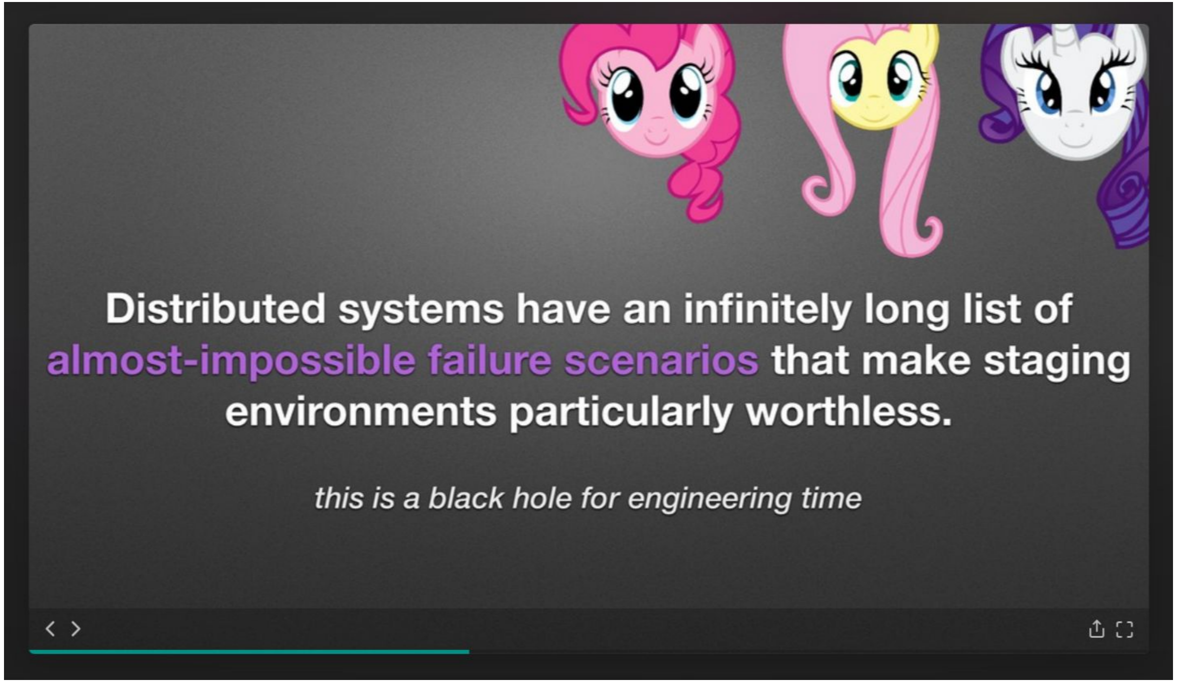
End to end functional testing of behaviours before users also see them.

Verifying config changes do not break those behaviours just as you do for code.

Useful especially if parts of configuration are in the user request.

Can never be equivalent to production.

Staging environments



https://speakerdeck.com/charity/engineering-large-systems-when-youre-not-google-or-facebook-test-in-prod?slide=12

Property-based Testing and Fuzzing

Recall our state space is the Cartesian product of the dimensions of our config C with a possibly large but finite number of values.

$$\prod_{i=1}^{n} |C_n|$$

Fuzz is a useful exploratory tool when the state space of the input is intractably large to brute force, and also fun.

Fuzzers don't test behaviour and don't know how to make logical tests, and can take a long time to uncover a bug

Progressive Rollouts

Pre-production testing cannot reach 100% coverage.

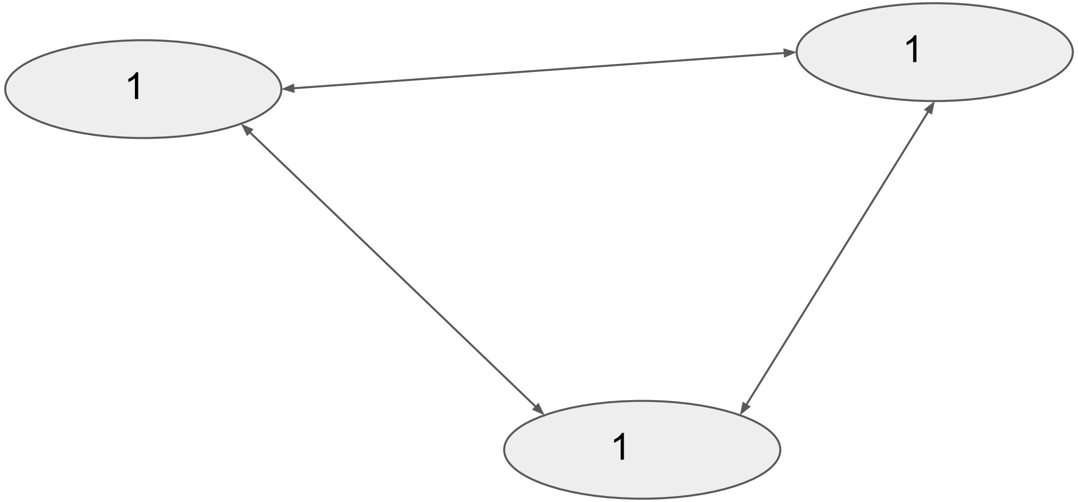
The final test for config changes are when it hits production.

The safest way to manage that risk is progressive rollouts.

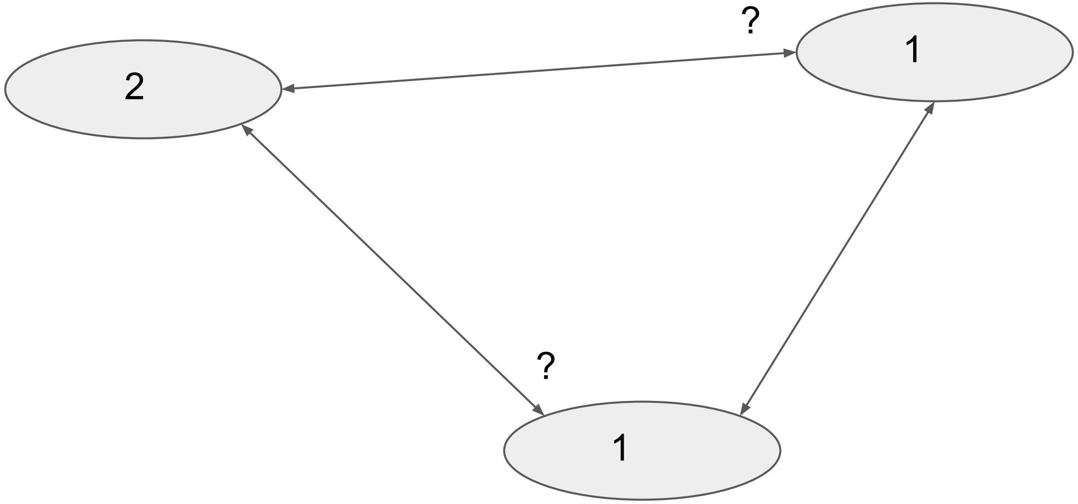
Bonus points for using automated analysis and stopping/rolling back if necessary.

Requires careful engineering of the system as well as the rollout system, and regular drilling on fast rollbacks

Progressive Rollouts and Split Brains



Progressive Rollouts and Split Brains



Progressive Rollouts of Config and the Split Brain

Some global systems pass messages between zones about their state, and make assumptions about those peers state.

During a config rollout, a peer might detect another is misbehaving or broken when it is really a change of parameters not visible yet to that peer.

One method to address this is to share local decision outputs as well as inputs in messages so the peers can crosscheck the work.

Delete code to simplify config

Simplicity is hard work, but things to look for:

- obsolete config flags never set, or set to defaults. Delete the condition and the path never taken.
    - Automate it! (ClangMR, go fix, etc)
- Machine-discoverable information. Instead of passing task counts in a config, and needing to keep that synchronised, let the program query for task counts. Opportunity allows for autoscaling.
    - Example: GCS backend
- Stop Config Spirals, remove scaffolding, layers of abstraction (hark back to *aiding understanding*)

Change the power of your config language

- Low power languages are less likely to have unexpected side effects inside their own scope.
    - Power reduction allows automation to make more assumptions and optimisations about intent.
    - Requires more options to capture the nuance of user intent.
    - Language modification is a small barrier to change
- High power languages are more likely to be able to describe the user's intent correctly.
    - Power increase reduces ability for machines to understand intent
    - Requires less work on part of implementer to capture all possible meanings and allows users to adapt to change.
    - Use an existing popular languages improves operator understanding
- Both directions lead to outages!

Other mitigations (TODO EXPAND)

- progressive rollouts of config, testing in prod, canary analysis
- pre-submit validation using same code as production in smaller binaries
- code formatters, and generated config differs
- fuzzing. earlier observation that config is a large state space; we can explore with fuzzers.
- actual functional testing, that’s what staging environments are for
- continuous integration of configs as well as code.
- deleting of config paths when no longer used; clang-mr; reference the Mythical Man Month on “simplicity”, but advice on what to look for (ala blobstore config)
- mitigations for split brain when doing progressive rollouts (ala autopilot global config)

Configuration is Code, treat it so

1. In theory, configuration should be a high risk of outage
2. Experimentally, not enough data to make strong conclusion
3. Risk mitigations that treat config the same as code work very well
    a. Everything in version control and no out-of-band maintenance
    b. Continuous testing and high coverage
    c. Fast feedback
    d. Contextful error messages
    e. Safe rollout practices and fast rollbacks
    f. Delete everything you don’t need
    g. Automate it

References!

You can find links to the external references by visiting the URL encoded in this QR Code.

Also please rate this session on the O'Reilly website or mobile app by clicking the big yellow button on the session page.