PETER ALVARO

The Twilight of the Experts

With a prelude of myths

and an appendix of dreams

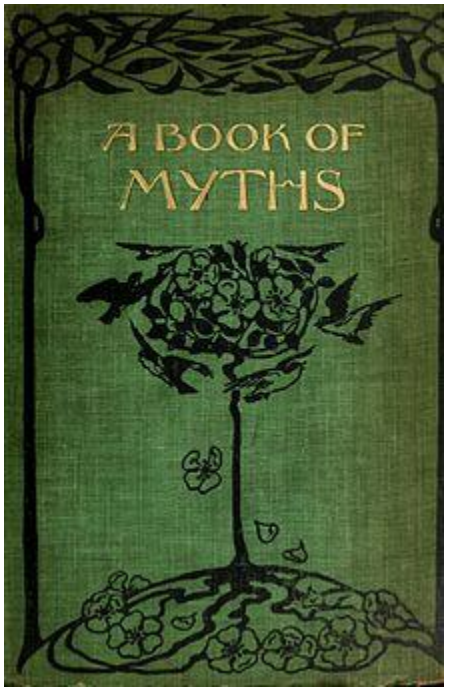
Distributed systems are hard

Your distributed system will suffer partial failures.

So it was built to tolerate them.

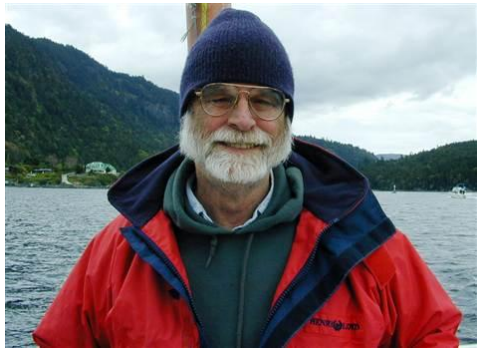
Will it, though?

Some old myths and a new one

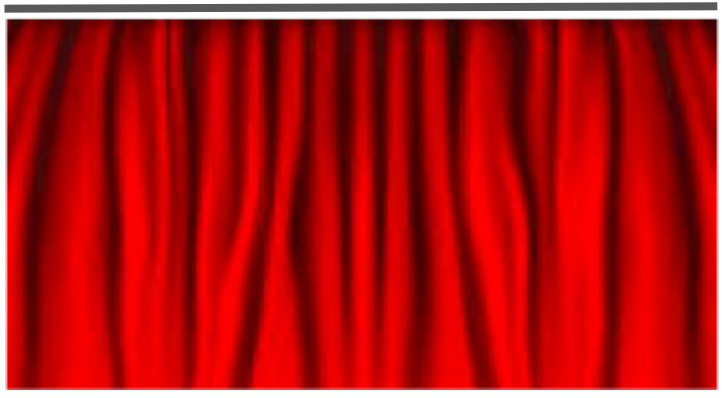


The old gods

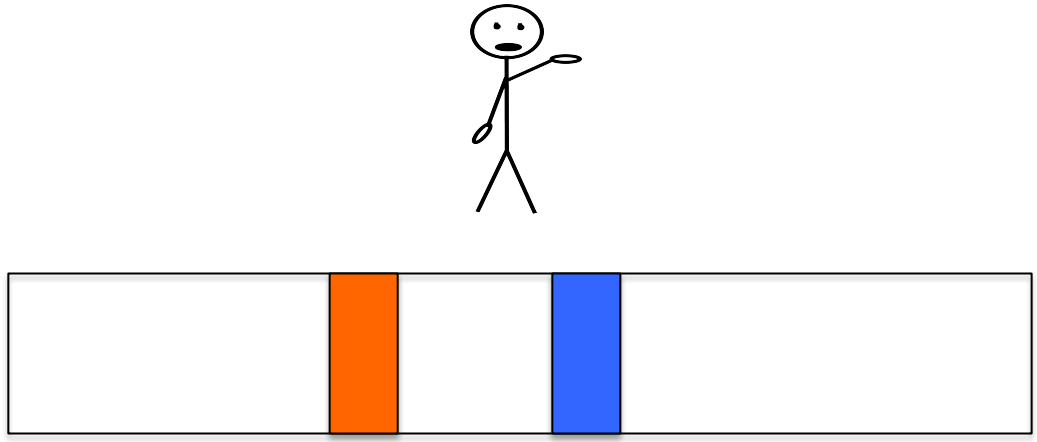
The ancient myth: *leave it to the experts*



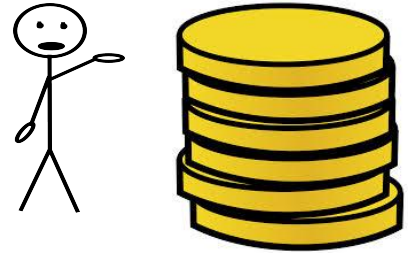
Fault tolerance via experts + abstraction



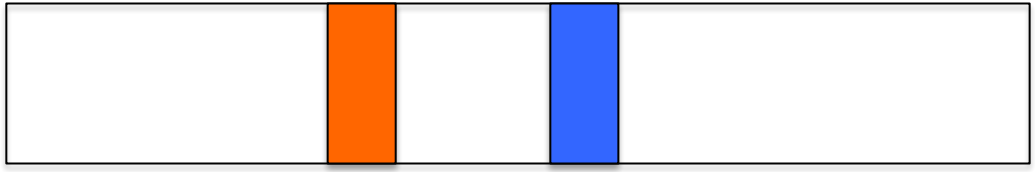
Example: RAID



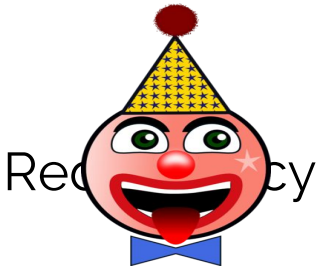
Example: RAID



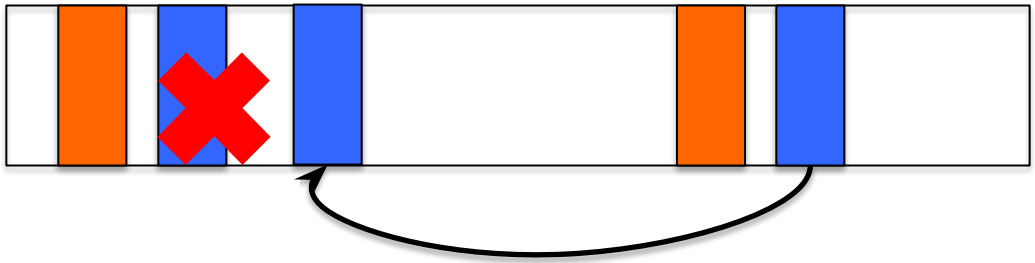
Transparency



Logical volumes

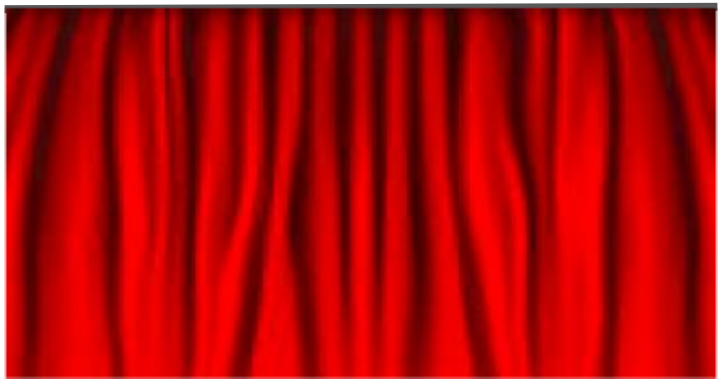


Rec cy

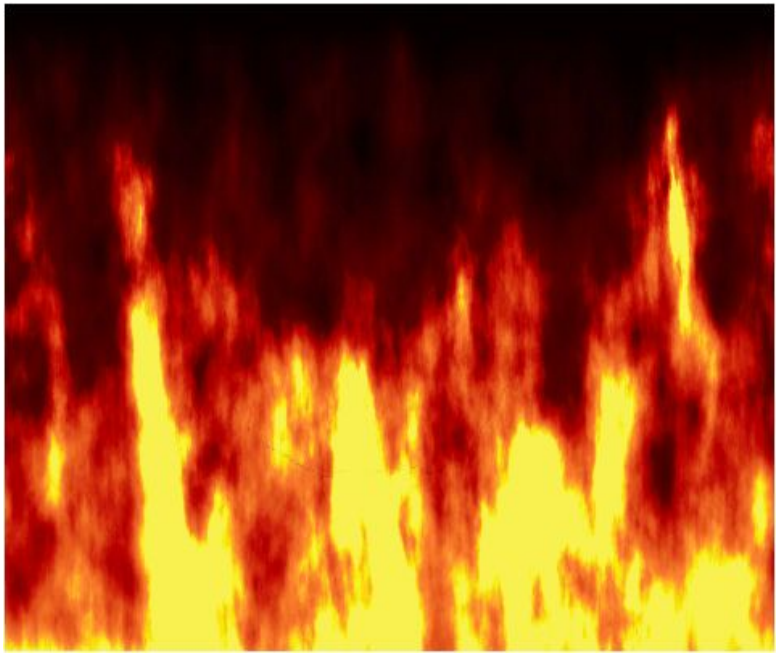


Physical volumes

Fault tolerance via experts + abstraction

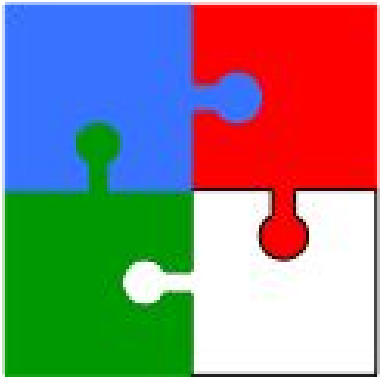


Fault tolerance via experts + abstraction



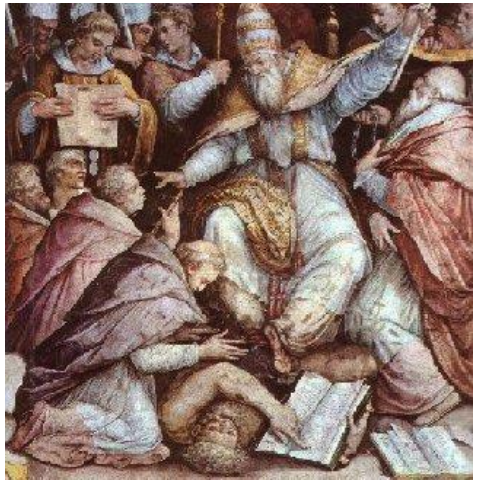
A fact

Composition is hard.



A secret

I’m honestly not sure what to do about it.

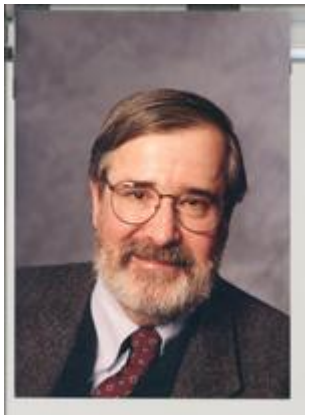


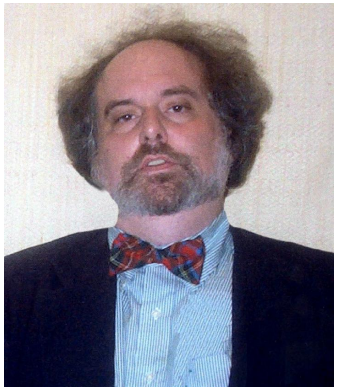
An opinion

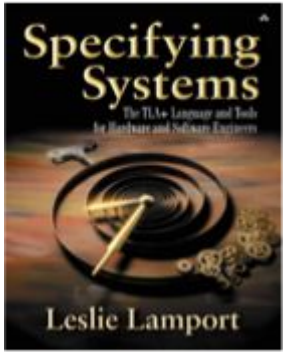
To hell with tech priesthoods

The old guard

The modern myth: *formally-verified distributed components*







A shift

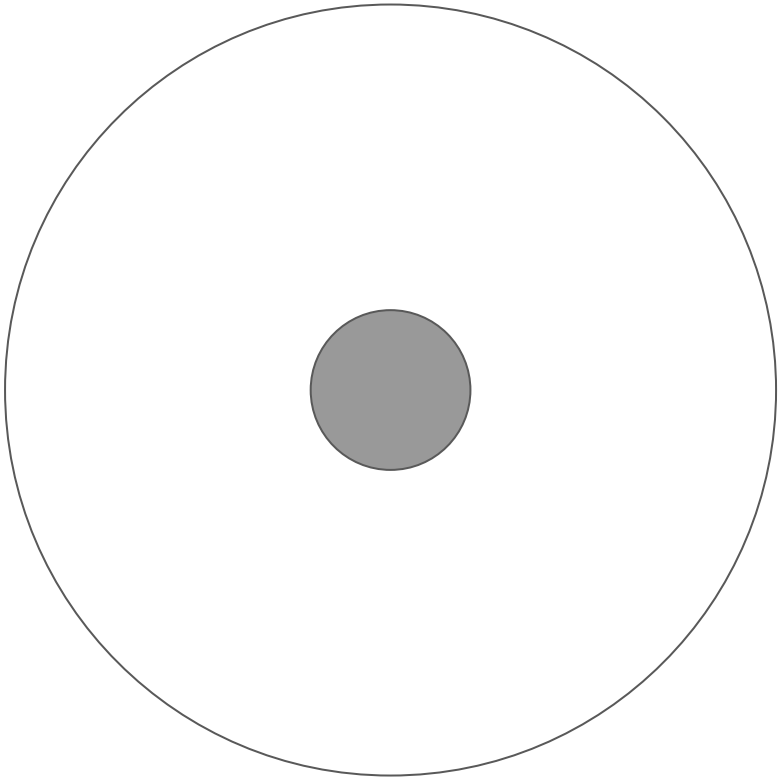
Formal methods

Testing +
Fault injection

Testing: walking around the black box

“Depth” of bugs

Single Faults



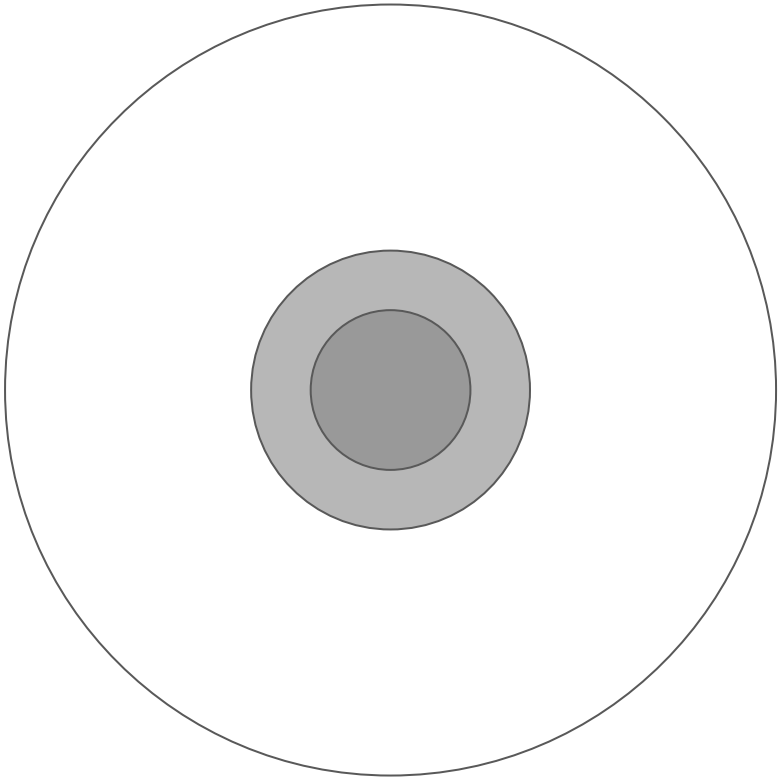
Search Space:
100 executions

Consider computation
involving 100 services

“Depth” of bugs

Combination of 4 faults

Consider computation involving 100 services

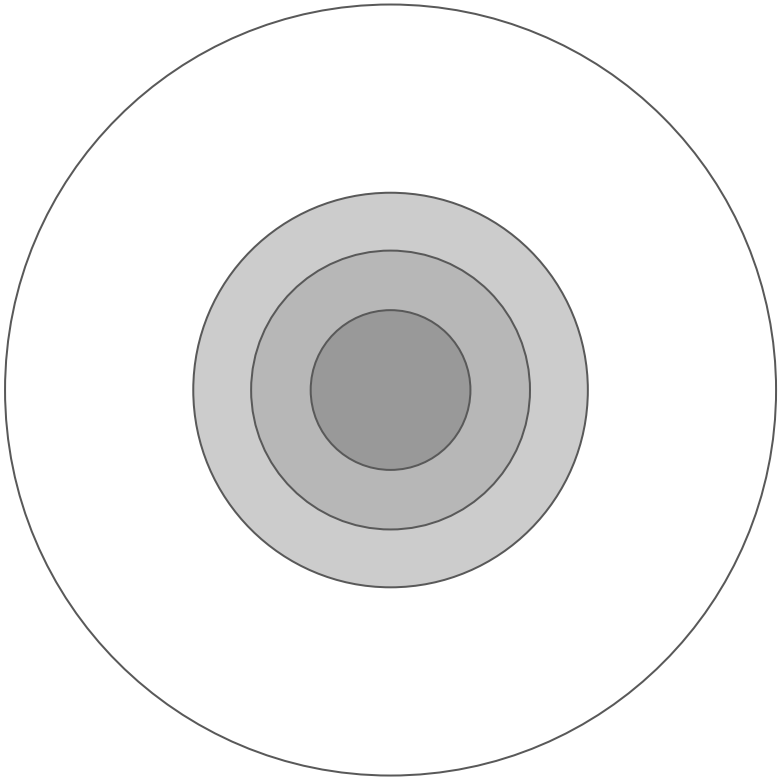


Search Space:
3M executions

“Depth” of bugs

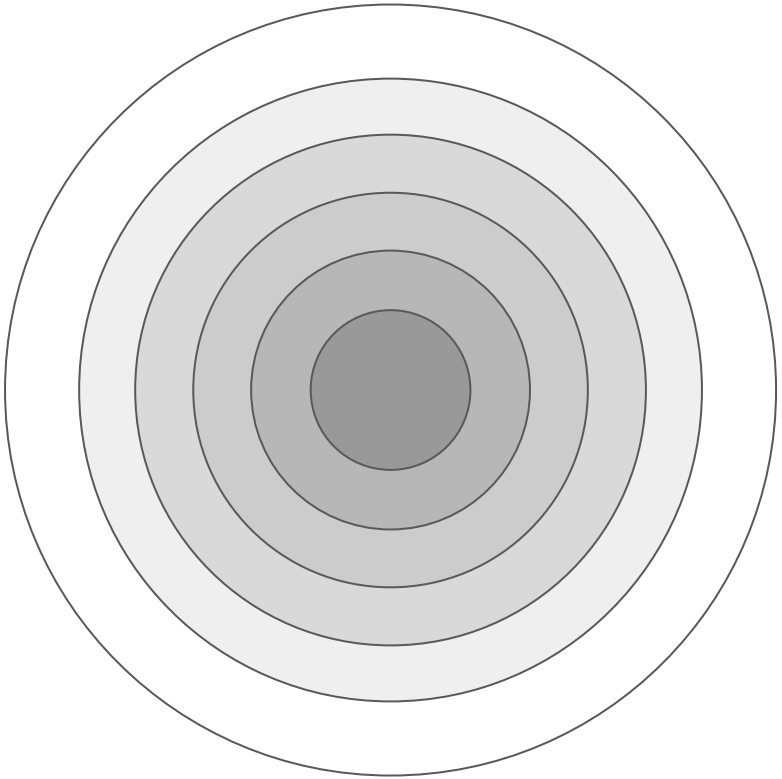
Combination of 7 faults

Consider computation involving 100 services



Search Space:
16B executions

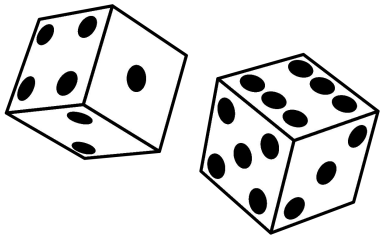
What could *possibly* go wrong?

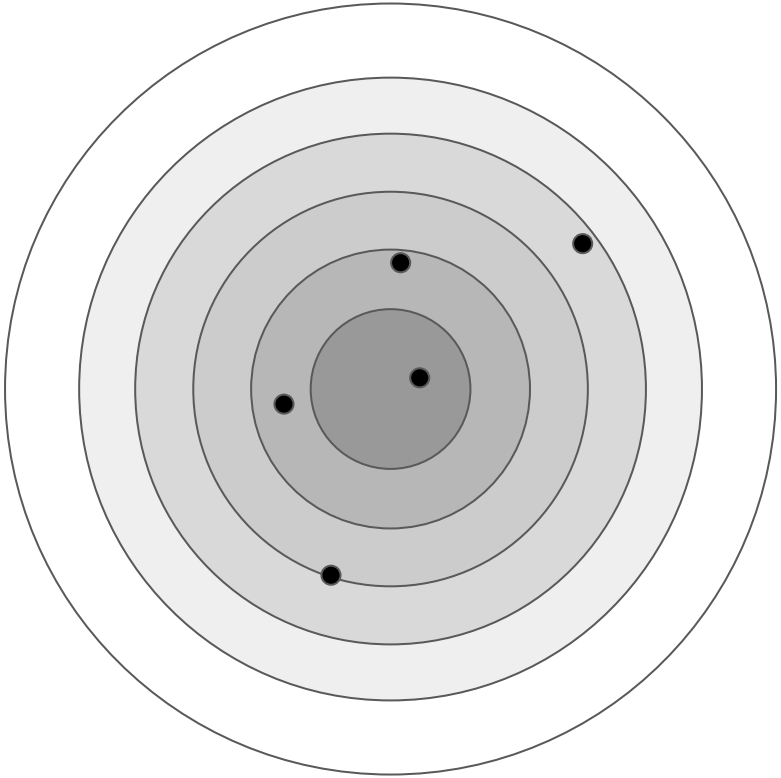


Search Space:
2^{100} executions

Consider computation
involving 100 services

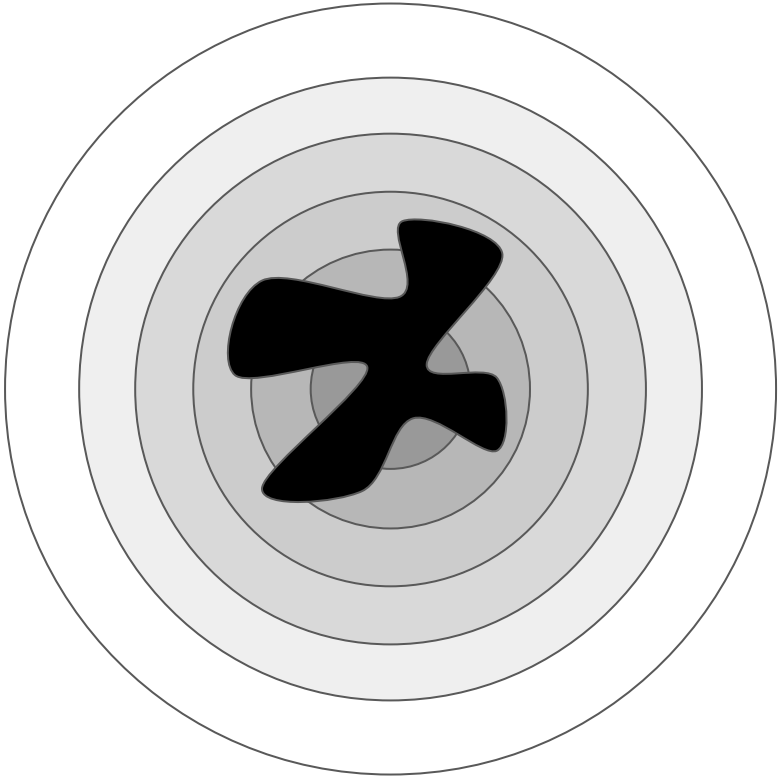
Random Search





Search Space:
2^{100} executions

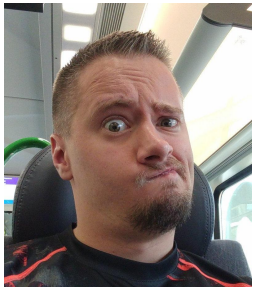
The vanguard: genius-guided Search



Search Space:
???

The vanguard

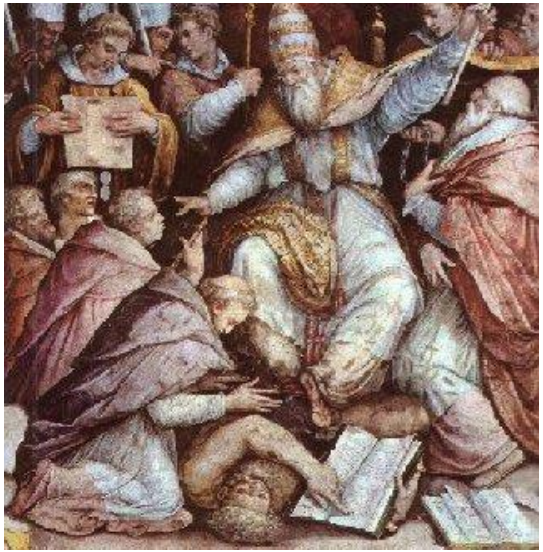
Chaos Engineering	Jepsen Testing
Web-scale applications	Distributed databases
10000s+ of machines	10s of machines
Polyglot	Often closed-source
Availability is king	Correctness is king



Down with the priesthoods

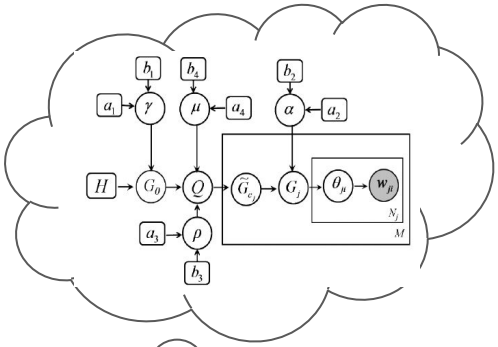
A problem: *experts are rare and expensive. Superusers are one-of-a-kind.*

A conjecture: *we can imitate the best practices of experts in software. Here's how.*



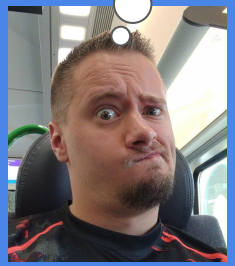
How do the experts do it?

Smarties #1: Jepsen testing



Read docs

Observe executions



Targeted faults

Observe **Think** **Act**

Smarties #2: Chaos engineering

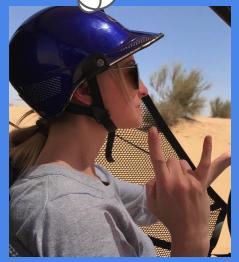
$$m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + kx + \mu(x - y) = f_o$$

$$\lambda\frac{d(y-z)}{dt} + \mu(y - x) = 0$$

$$M\frac{d^2z}{dt^2} + B\frac{dz}{dt} + \lambda\frac{d(z-y)}{dt} + Kz = f_s$$

Meet teams

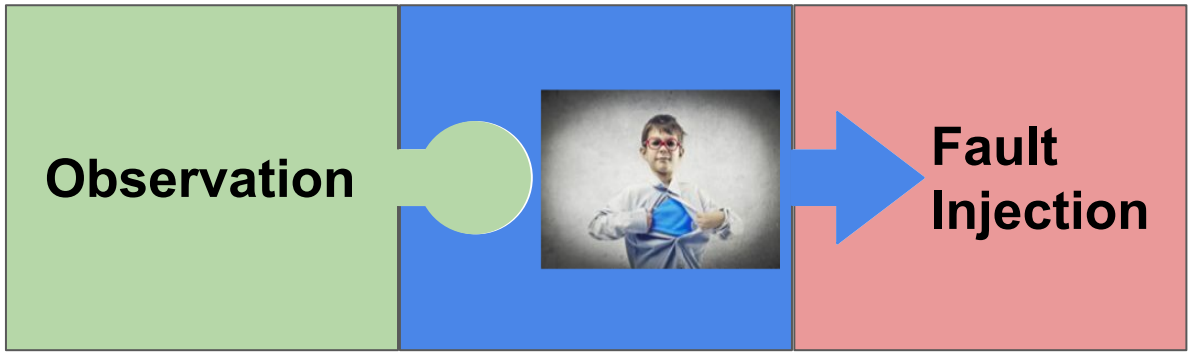
Observe executions



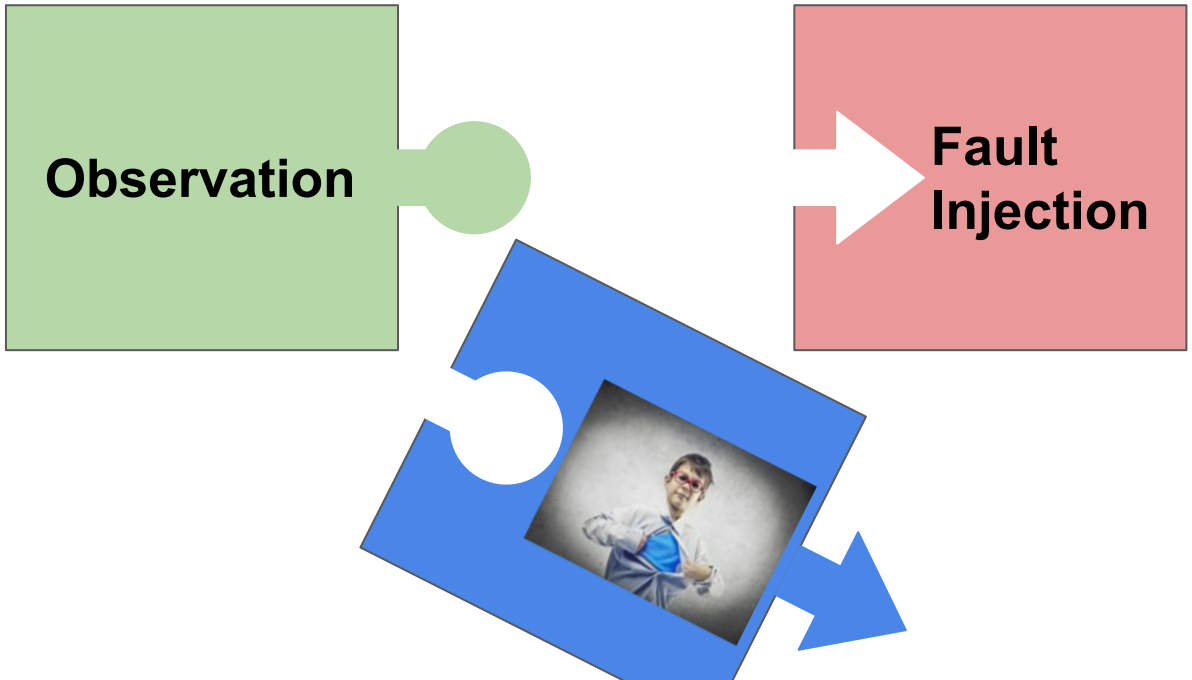
Targeted faults

Observe **Think** **Act**

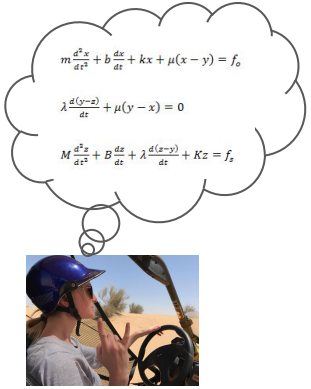
The genius in the loop



The genius in the loop



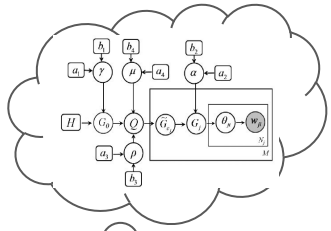
How do the experts do it?

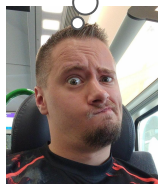


Observation

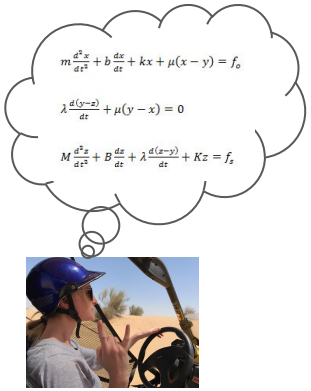
A mental model

Fault Injection





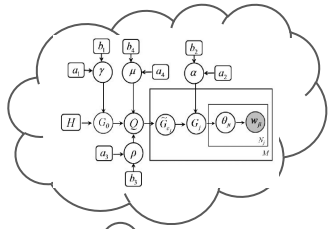
How do the experts do it?

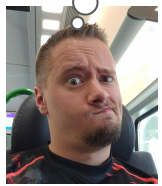


Observation

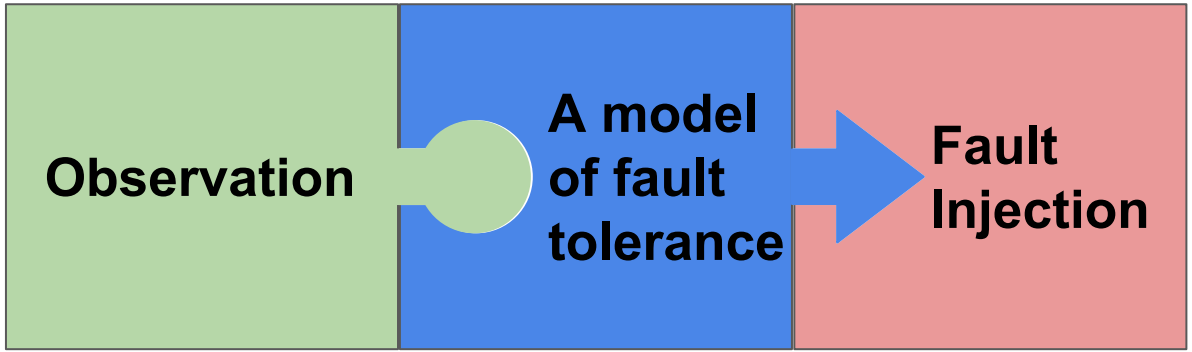
A mental model

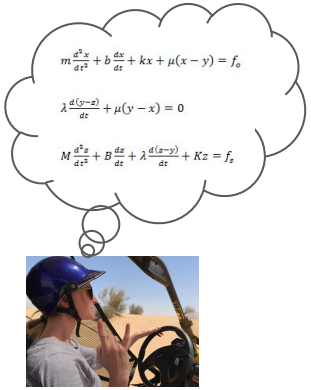
Fault Injection

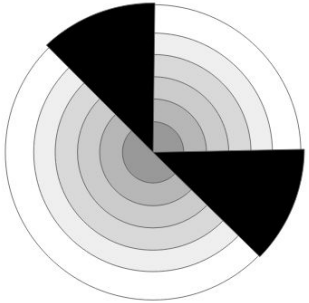


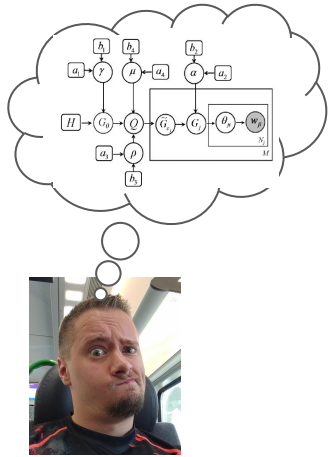


How do the experts do it?

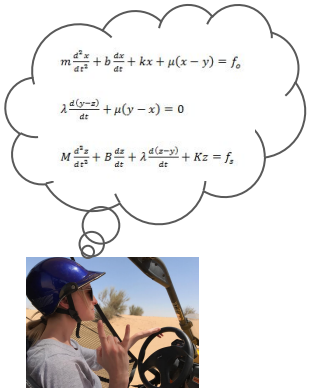








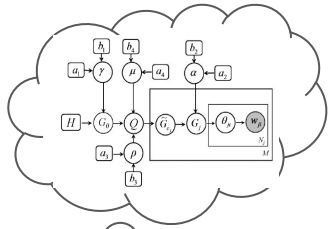
How do the experts do it?

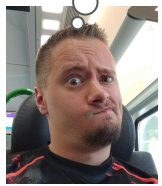


Observation

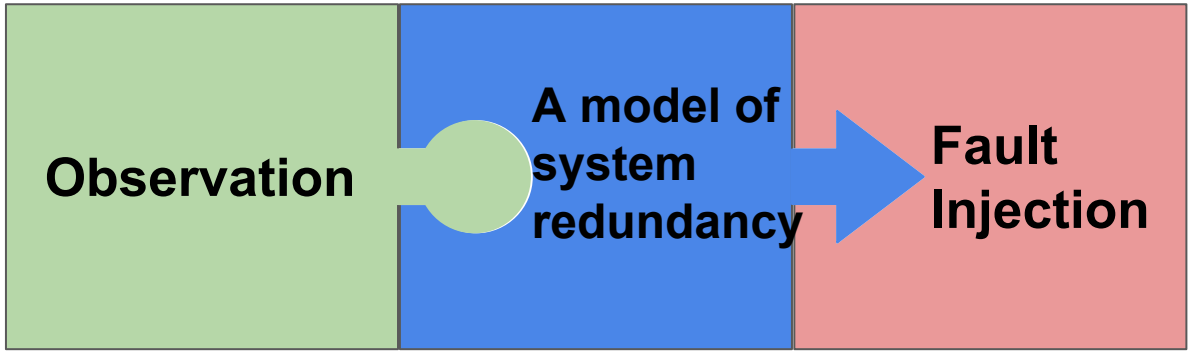
A model of fault tolerance

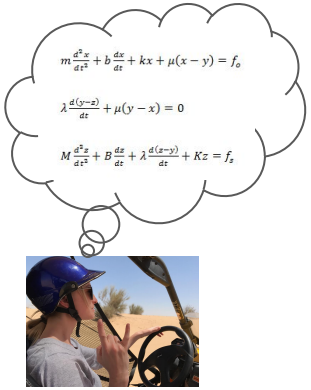
Fault Injection

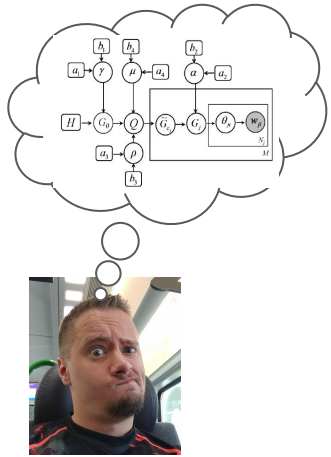




How do the experts do it?







A proof by construction

Lineage-driven Fault Injection

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joshua Rosen
UC Berkeley
rosenville@gmail.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

We propose a novel approach for discovering bugs in fault-tolerant data management systems: *lineage-driven fault injection*. A lineage-driven fault injector reasons *backwards* from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

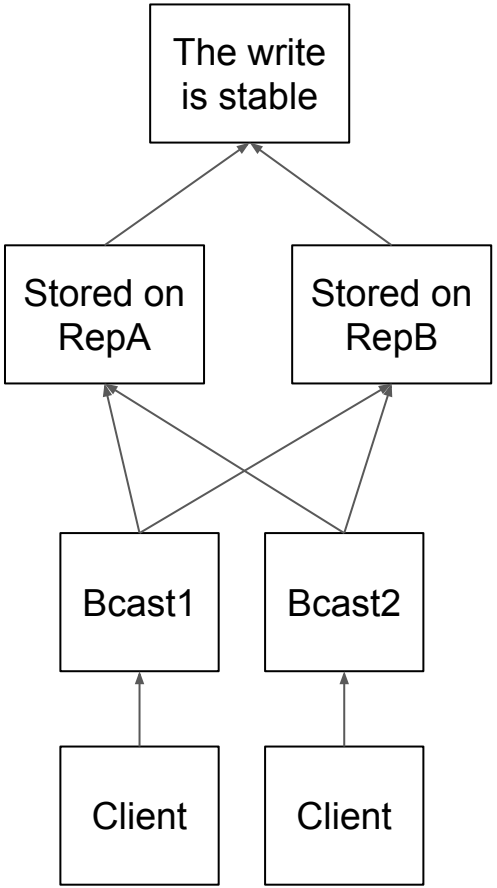
enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 57] and bugs [36, 49].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection [1, 26, 36, 44, 59] is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure). Ap-

Lineage-driven fault injection

Why did a good thing happen?

Consider its *lineage*.



Lineage-driven fault injection

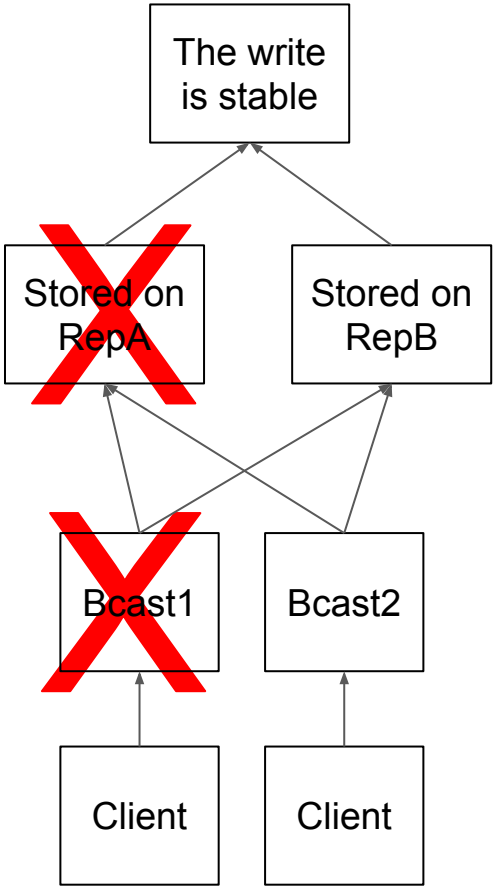
Why did a good thing happen?

Consider its *lineage*.

What could have gone wrong?

Faults are *cuts* in the lineage graph.

Is there a cut that breaks all supports?



Lineage-driven fault injection

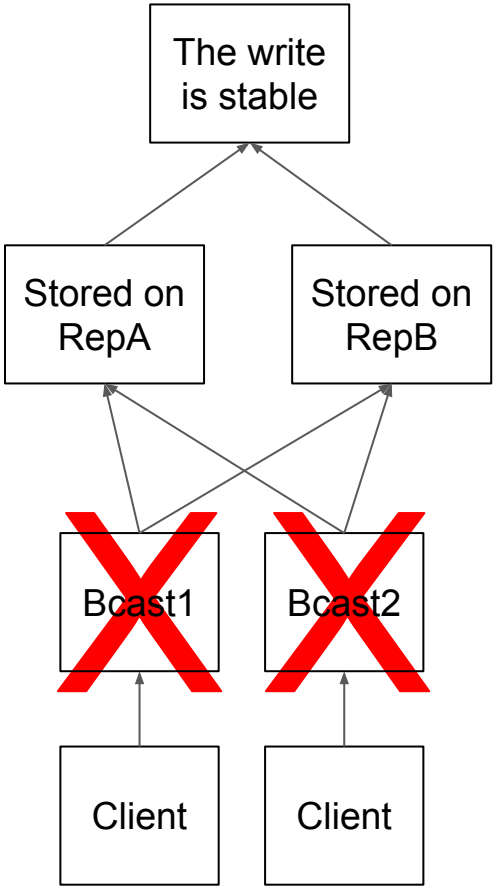
Why did a good thing happen?

Consider its *lineage*.

What could have gone wrong?

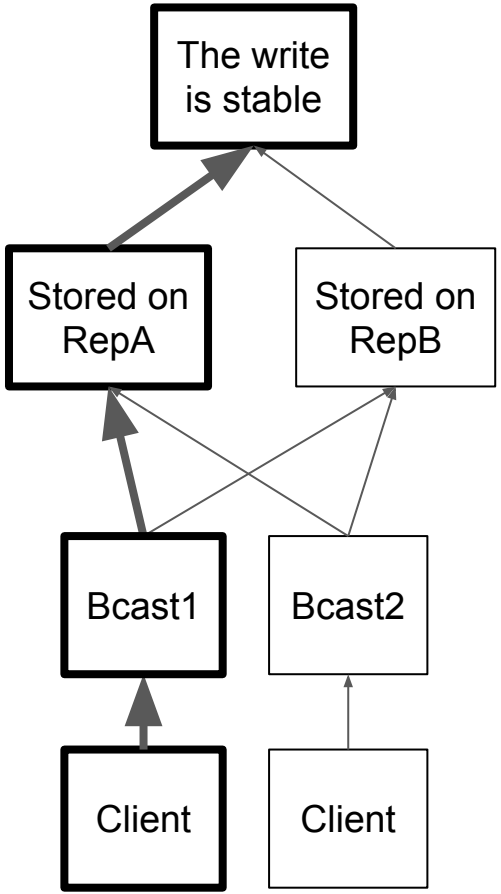
Faults are *cuts* in the lineage graph.

Is there a cut that breaks all supports?



What would *have to* go wrong?

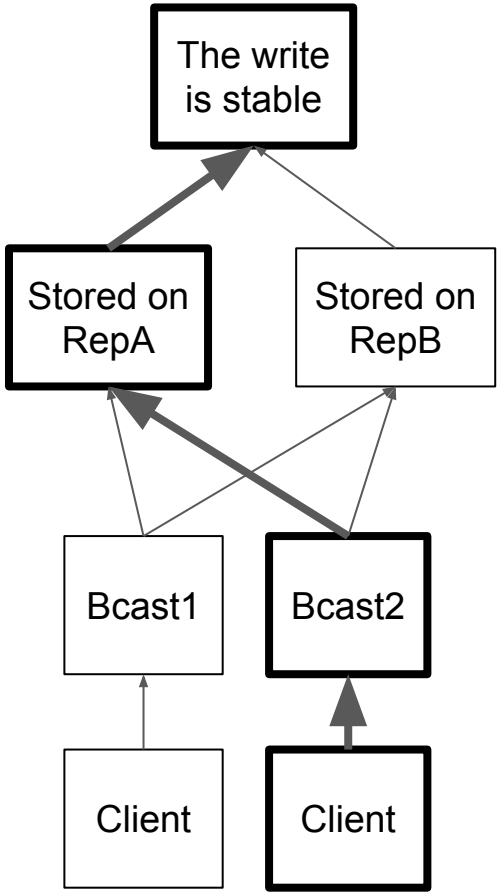
(RepA OR Bcast1)



What would *have to* go wrong?

(RepA OR Bcast1)

AND (RepA OR Bcast2)

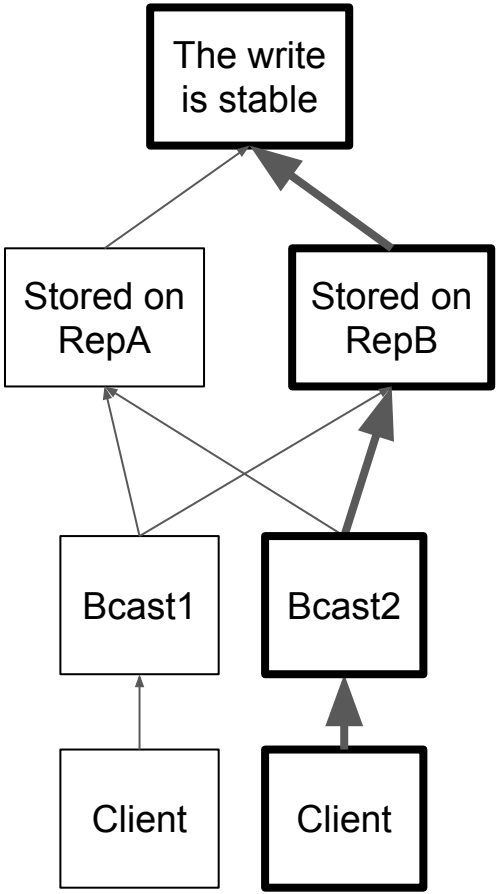


What would *have to* go wrong?

(RepA OR Bcast1)

AND (RepA OR Bcast2)

AND (RepB OR Bcast2)



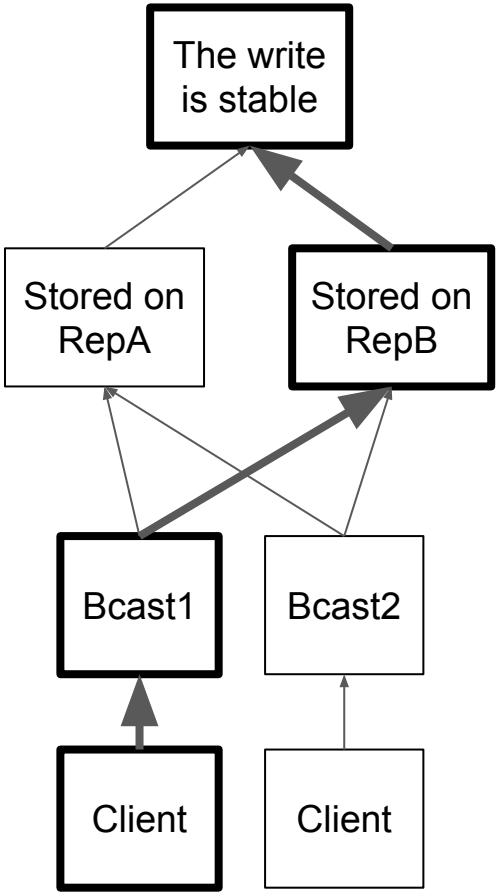
What would *have to* go wrong?

(RepA OR Bcast1)

AND (RepA OR Bcast2)

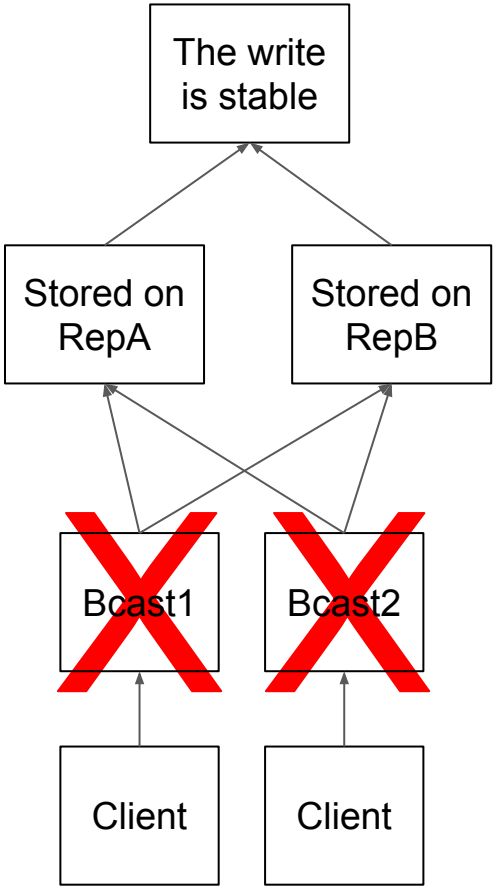
AND (RepB OR Bcast2)

AND (RepB OR Bcast1)



Lineage-driven fault injection

Hypothesis: {Bcast1, Bcast2}



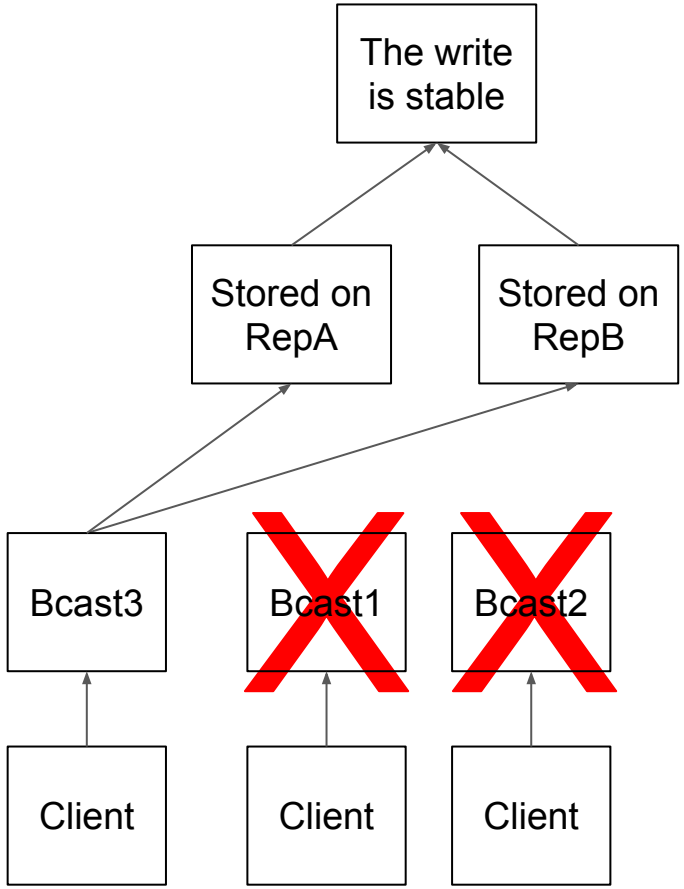
Lineage-driven fault injection

(RepA OR Bcast1)

AND (RepA OR Bcast2)

AND (RepB OR Bcast2)

AND (RepB OR Bcast1)



Lineage-driven fault injection

(RepA OR Bcast1)

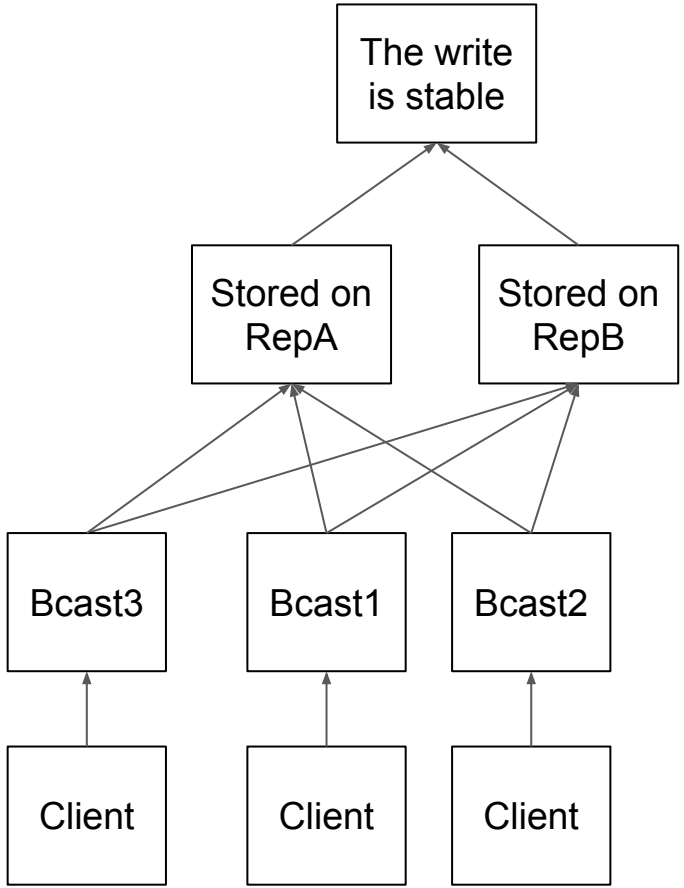
AND (RepA OR Bcast2)

AND (RepB OR Bcast2)

AND (RepB OR Bcast1)

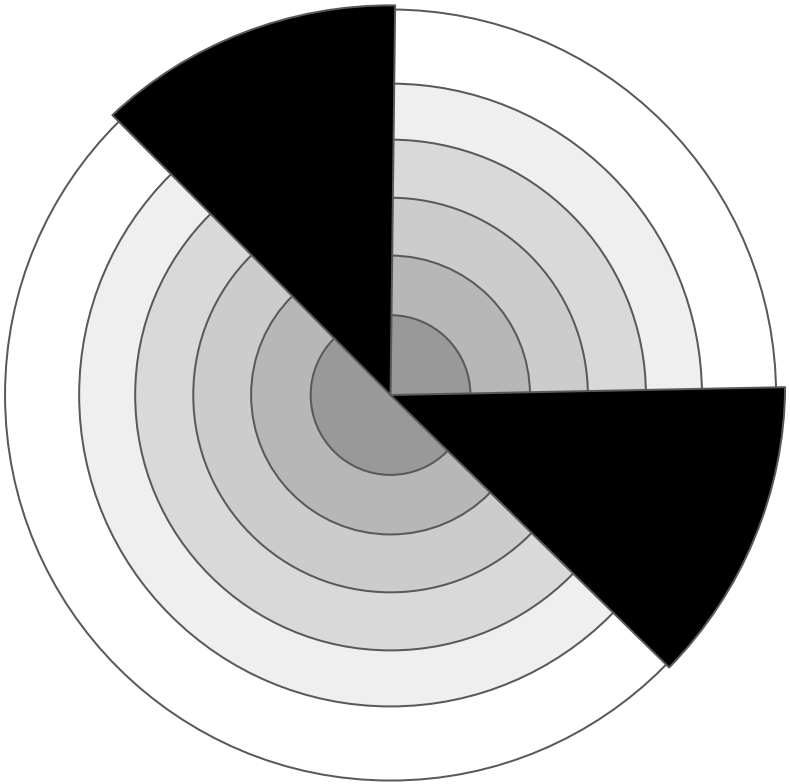
AND (RepA OR Bcast3)

AND (RepB OR Bcast3)



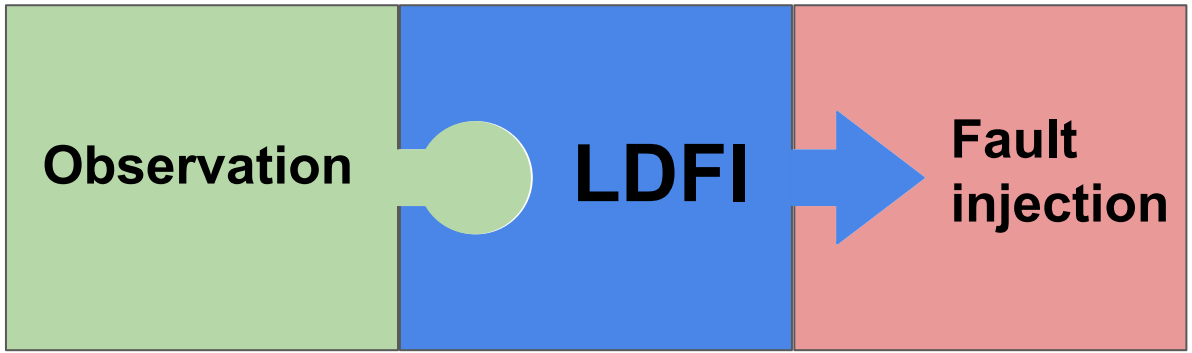
Search Space Reduction

Each Experiment finds a bug, OR



Reduces the Search space

Lineage-driven Fault Injection



LDFI Successes

Finding bugs in protocols [**SIGMOD'15, HotCloud'17**]

Finding bugs in large-scale applications [**SoCC'16**]

Finding funding! [**NSF CAREER 2017-2021**]

kafka

elastic

NETFLIX

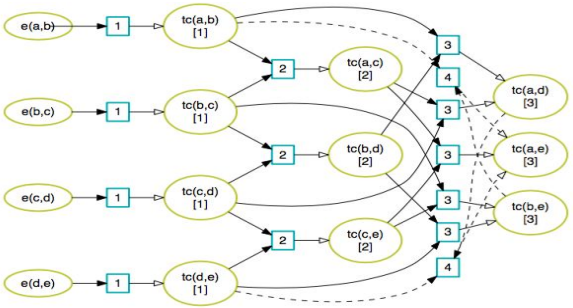
UBER

NSF

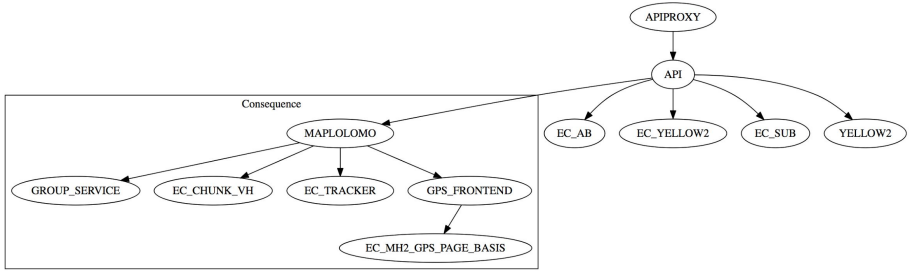
Some dreams

Explanations everywhere

Database provenance

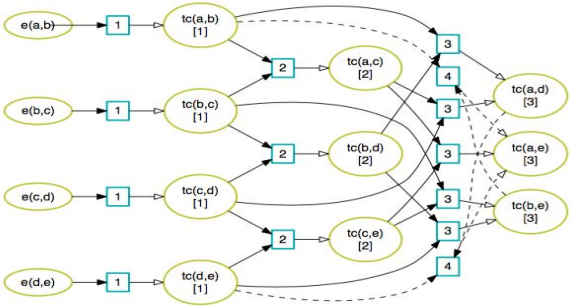


Call graphs

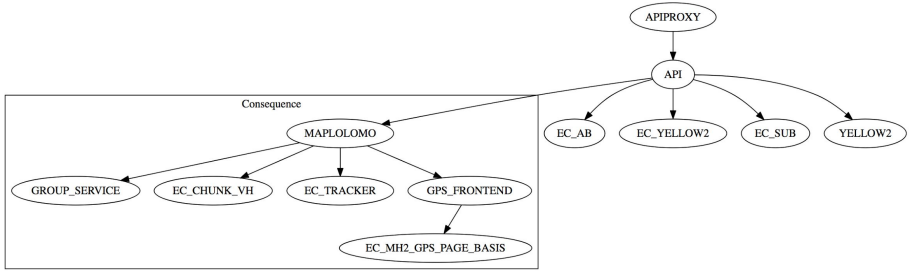


Explanations everywhere

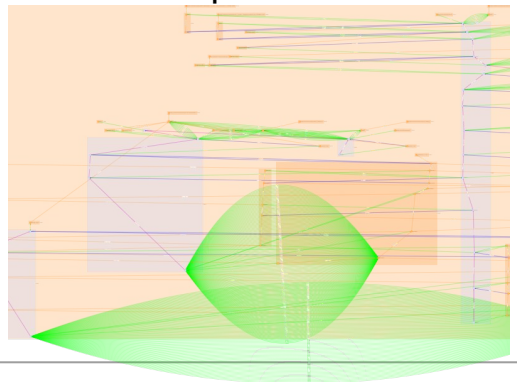
Database provenance



Call graphs



OS-level provenance?

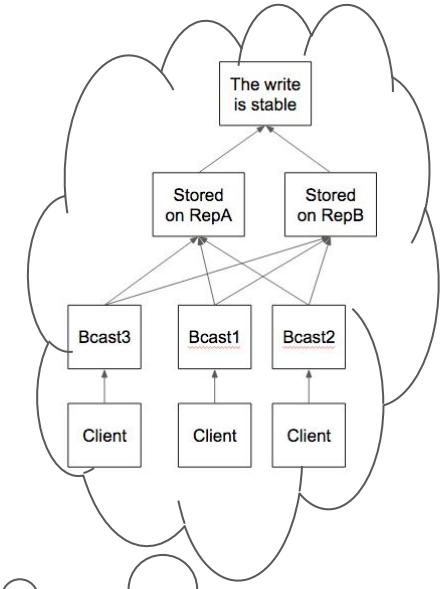


Unstructured logs?

?

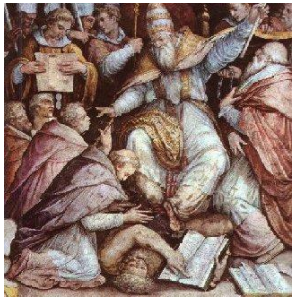
Towards better models

??



Remember

1. Composability is the last hard problem
2. To hell with priesthoods!
3. We can automate the peculiar genius of experts



Thanks to our hosts, benefactors and collaborators!

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Berkeley
UNIVERSITY OF CALIFORNIA

NETFLIX

elastic

GREMLIN

UBER

facebook

References

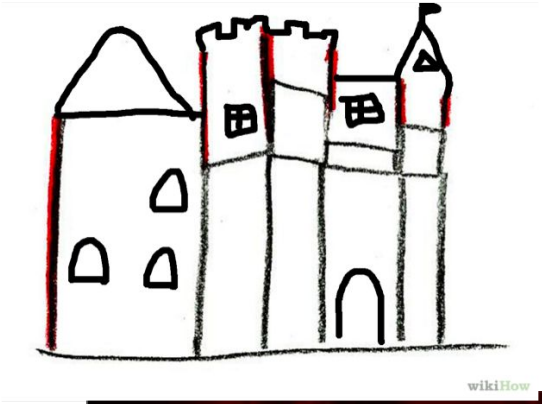
- ‘Automating Failure Testing at Internet Scale [**ACM SoCC’16**] https://people.ucsc.edu/~palvaro/fit-ldfi.pdf

- ‘Lineage Driven Fault Injection’ [**ACM SIGMOD’15**] http://people.ucsc.edu/~palvaro/molly.pdf

- Netflix Tech Blog on ‘Automated Failure Testing’ http://techblog.netflix.com/2016/01/automated-failure-testing.html

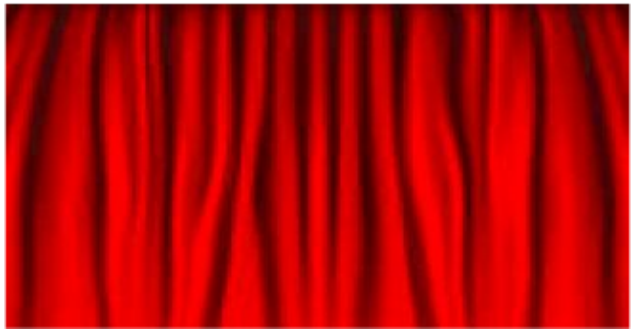
Alternative title: the circus animals' dissertation

(in which I reuse all of my old clip art from past talks)

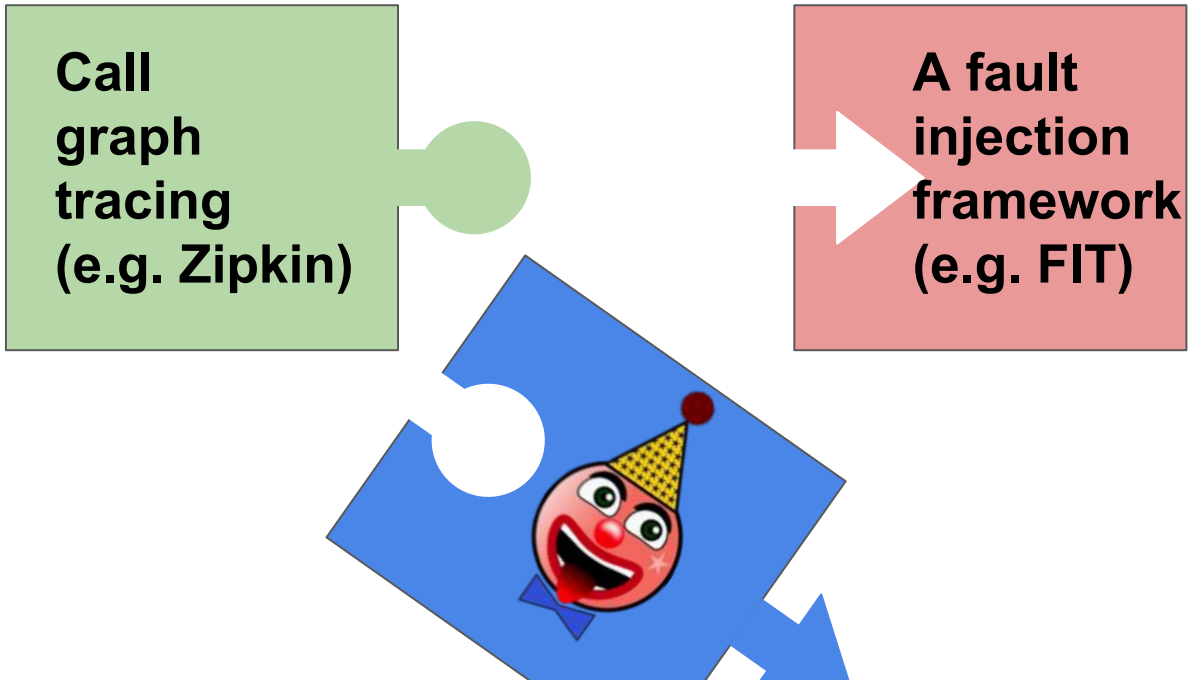


FOLD

Circus animals



A cunning malevolent sentience?



If you are targeting faults in a non-random way, you have a mental model of the system’s fault tolerance.

Fault tolerance is redundancy.

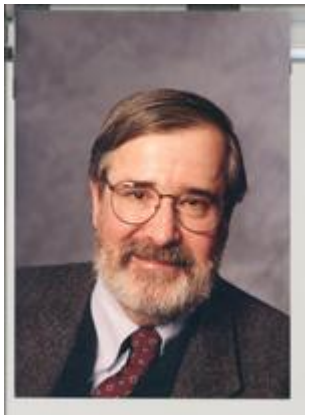
Hence your mental model is surely a model of a system’s redundancy.

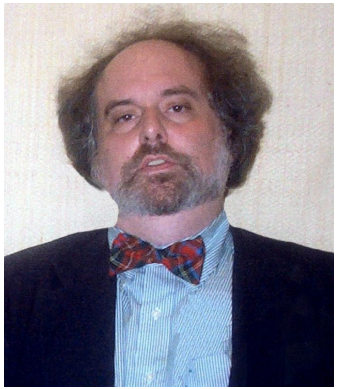
You have observed the system from the outside, so this model of redundancy must be built from observations of system behavior (under fault and not).

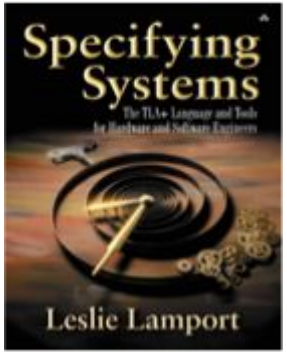
We can automatically build and maintain such models.

The old guard

The modern myth: *formally-verified distributed components*







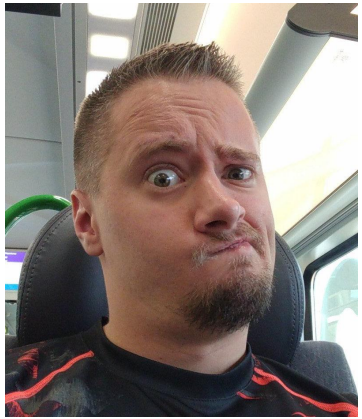
Eroding assumptions

1. ~~Experts~~
2. Specifications
3. Source code

The vanguard

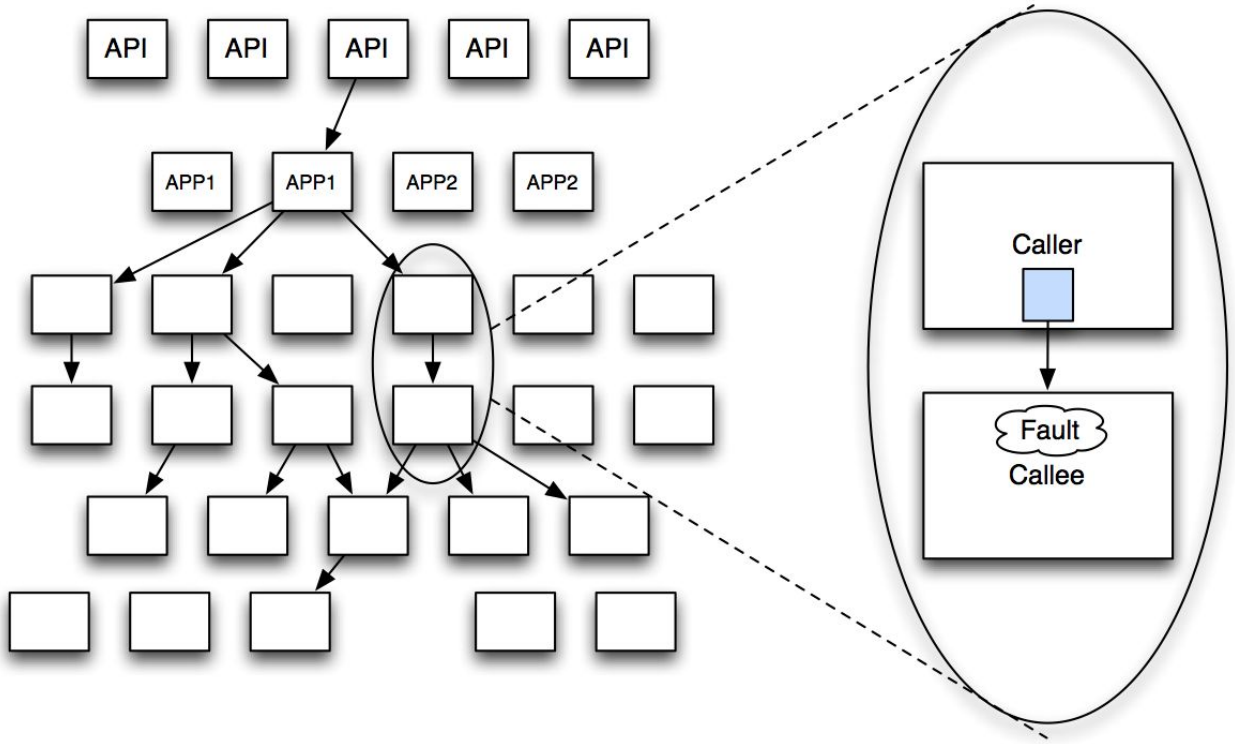
The emerging ethos: YOLO

Chaos Engineering



Jepsen Testing

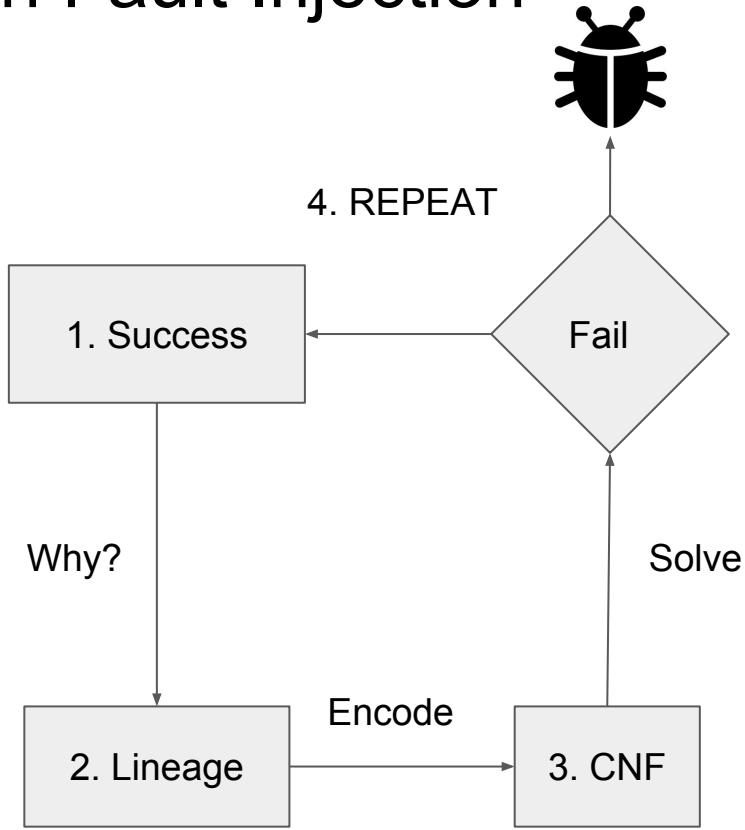
Don’t overthink fault injection



Lineage-driven Fault Injection

Recipe:

1. Start with a successful outcome. Work backwards.
2. Ask *why* it happened: Lineage
3. Convert lineage to a boolean formula and solve
4. Lather, rinse, repeat



How do we know redundancy when we see it?

Hard question: “Could a bad thing ever happen?”

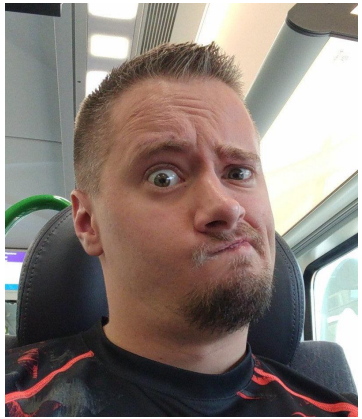
Easier: “Exactly *why* did a good thing happen?”

“What could have gone wrong?”

The vanguard

The emerging ethos: YOLO

Chaos Engineering



Jepsen Testing

Fault-tolerance “is just” redundancy

Lineage-driven Fault Injection

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joshua Rosen
UC Berkeley
rosenville@gmail.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

We propose a novel approach for discovering bugs in fault-tolerant data management systems: *lineage-driven fault injection*. A lineage-driven fault injector reasons *backwards* from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 57] and bugs [36, 49].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection [1, 26, 36, 44, 59] is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure). Ap-