



Observability and the Glorious Future

*What got you here won't get you there



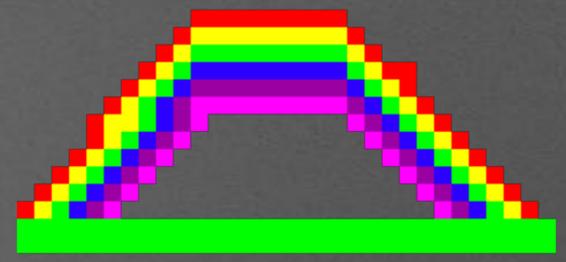
Creation myth

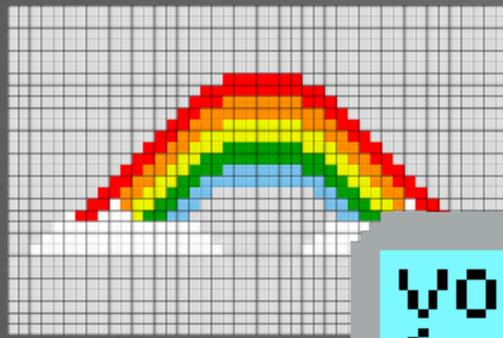


And engineers saw the code, and it was scary AF. So engineers then divided prod from staging, dev from ops, in a series of unfortunate and increasingly counterproductive solutions to the problem.









```
VOID  
{  
  CODECODE(CODE);  
}
```



write, deploy, debug



Software Ownership

They who **write** the code
Can and should **deploy** their code
And **support** it in production.

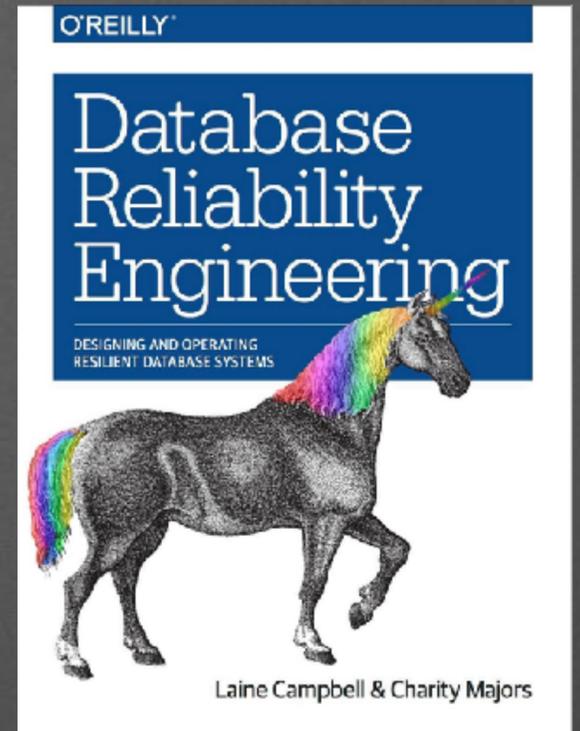


GOO





Observability



@mipsytipsy
engineer/cofounder/CEO

“the only good diff is a red diff”

<https://charity.wtf>





@mipsytipsy

Hates monitoring*

* Monitoring is very important!



Not a monitoring company

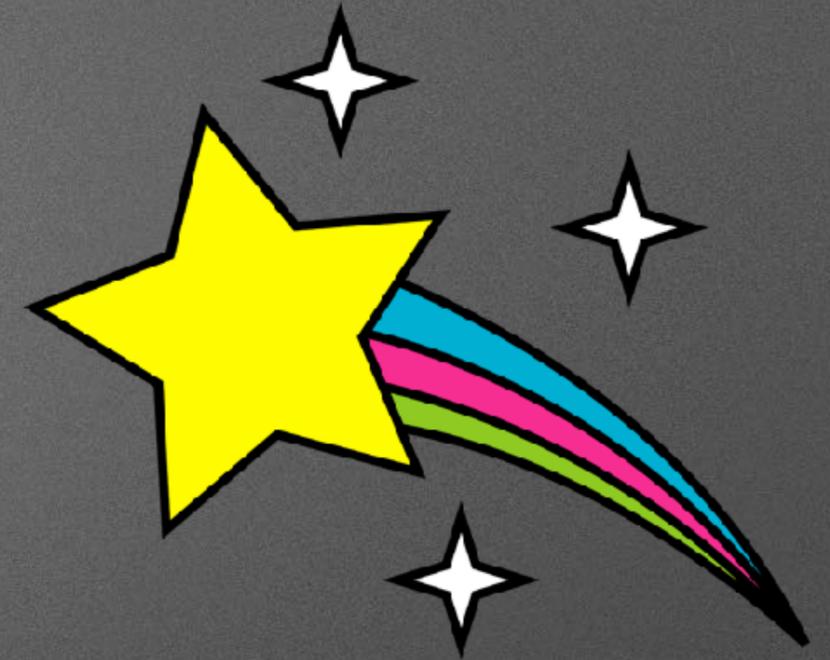
Thesis:

When healthy teams with good cultural values and leadership alignment try to adopt software ownership and fail, the cause is usually an observability gap.



Observability

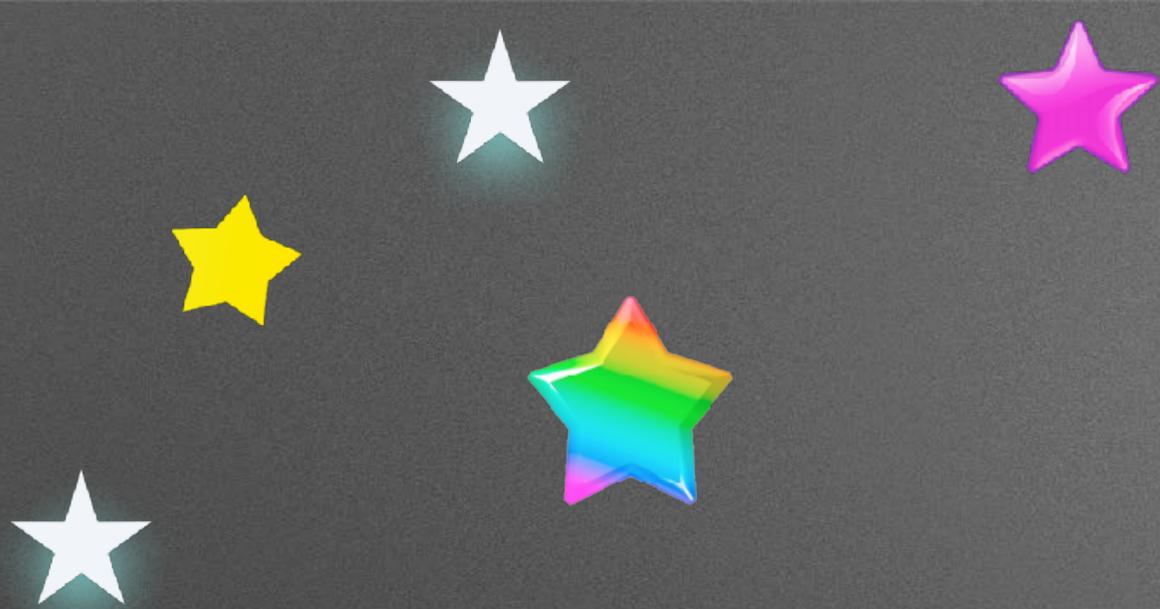




Observability

“In control theory, **observability** is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. The observability and controllability of a system are mathematical duals.” — wikipedia

... translate??!?



Observability for software engineers:

Can you understand what's happening inside your systems, just by asking questions from the outside? Can you debug your code and its behavior using its output?

Can you answer new questions **without shipping new code**?

You have an **observable** system
when your team can quickly and reliably track
down any **new** problem with no prior knowledge.

For software engineers, this means being able to
reason about your code, identify and fix bugs, and
understand user experiences and behaviors ...
via your instrumentation.





Monitoring

Represents the world from the perspective of a third party, and describes the health of the system and/or its components in aggregate.

Observability

Describes the world from the perspective of the software, as it performs each request. Software explaining itself back to you from the inside.



Monitoring

known-unknowns

"What's the capacity of the app tier, do I need to increase it? Is mean latency above historical norm? When will we run out of disk space?"

Observability

unknown-unknowns

"Why are we getting some complaints about users seeing the wrong image? Is the code I just wrote working as intended?"

We have been building observability tooling by and for operational teams for 20+ years.

Asking software engineers to own their code using our tools is asking them to have two full time jobs.

... so why now?

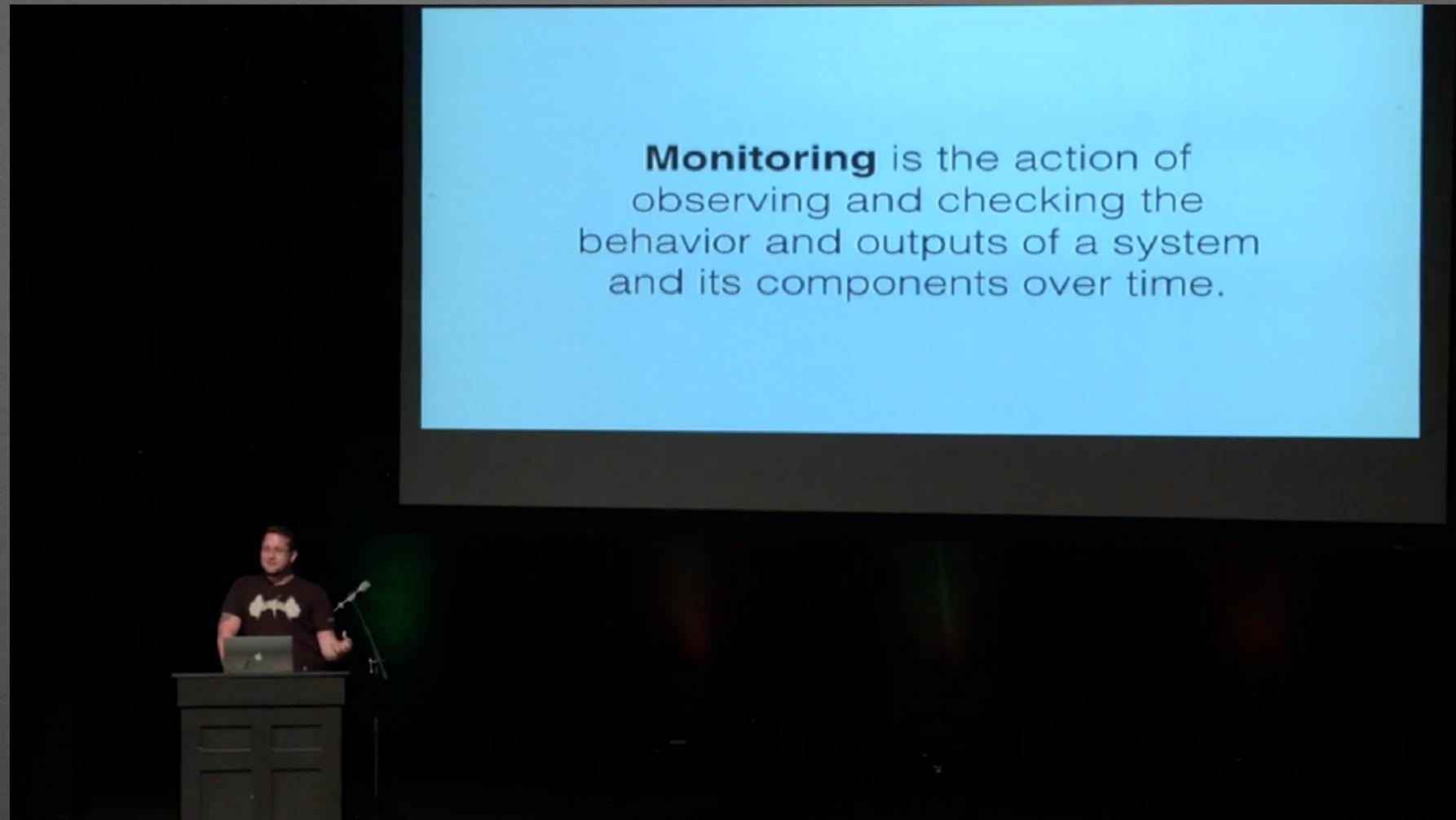




“Monitoring is dead.”

“Monitoring systems have not changed significantly in 20 years and has fallen behind the way we build software. Our software is now large distributed systems made up of many non-uniform interacting components while the core functionality of monitoring systems has stagnated.”

@gropory, Monitorama 2016



@grepory, 2016

This is a outdated model for complex systems.

We don't *know* what the questions are, all we have are unreliable symptoms or reports.

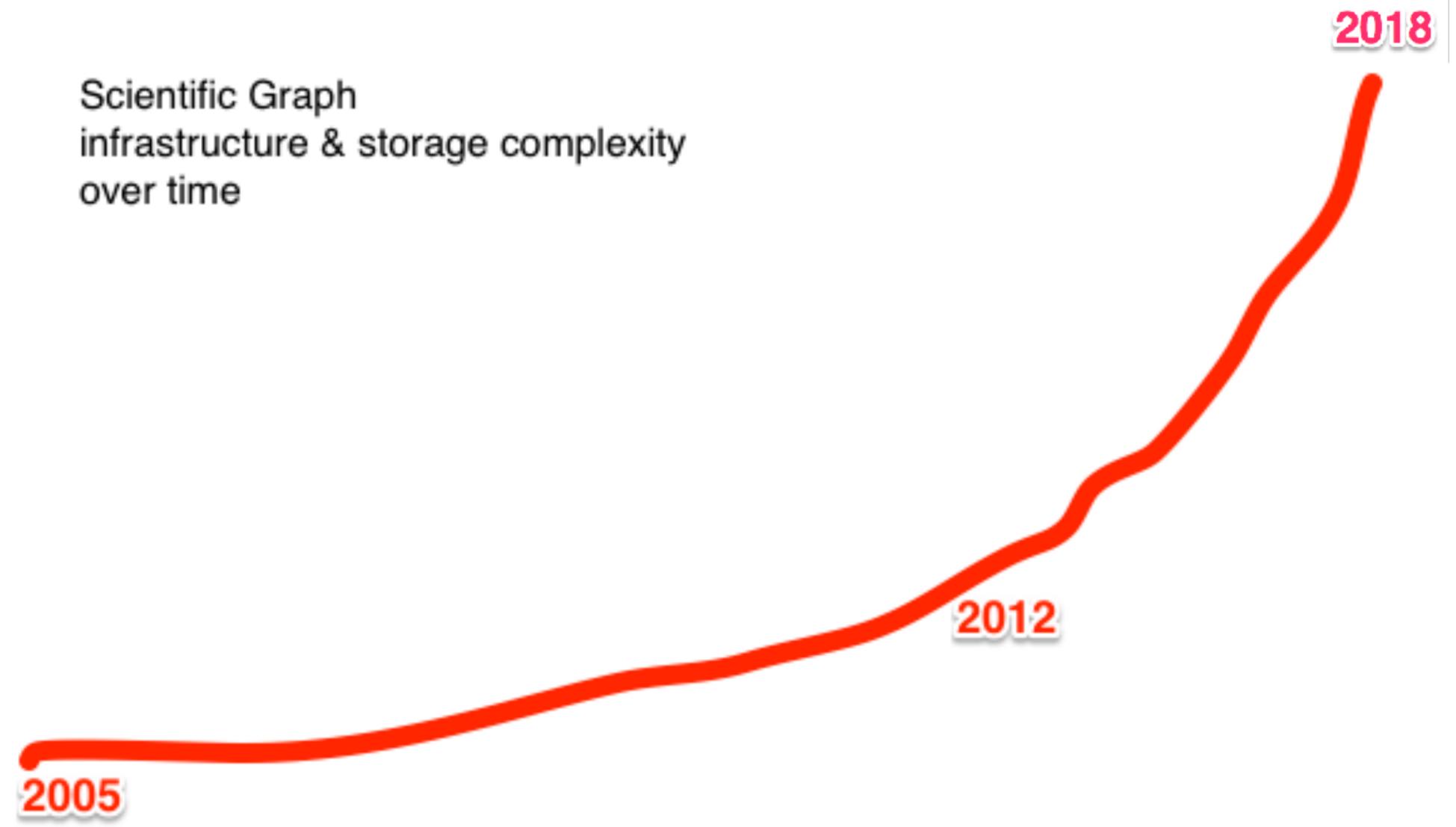
**Complexity is exploding everywhere,
but our tools are designed for
a predictable world.**



As soon as we know the question, we usually know the answer too.



Scientific Graph
infrastructure & storage complexity
over time

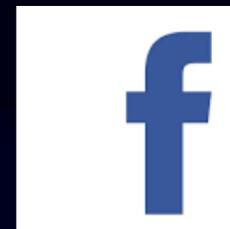
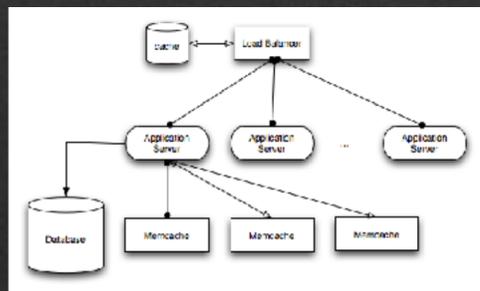
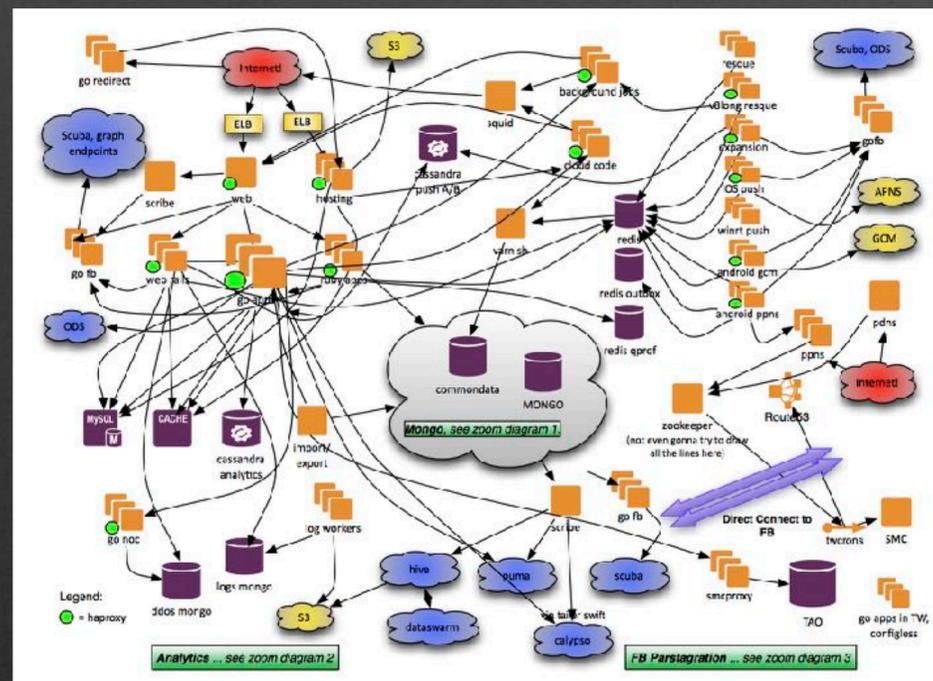


“Complexity is increasing” - Science



monitoring => observability

known unknowns => unknown unknowns



You're not bad at debugging,
your tools were designed for different problems



converging trends:

monolith => **microservices**

“the database” => **polyglot persistence**

users => **developers**

single tenant => **multi tenancy app**

could reason about => **def cannot reason about**

distributed systems:

it is often harder to find out where the problem is, than what the problem is.

Welcome to distributed systems.

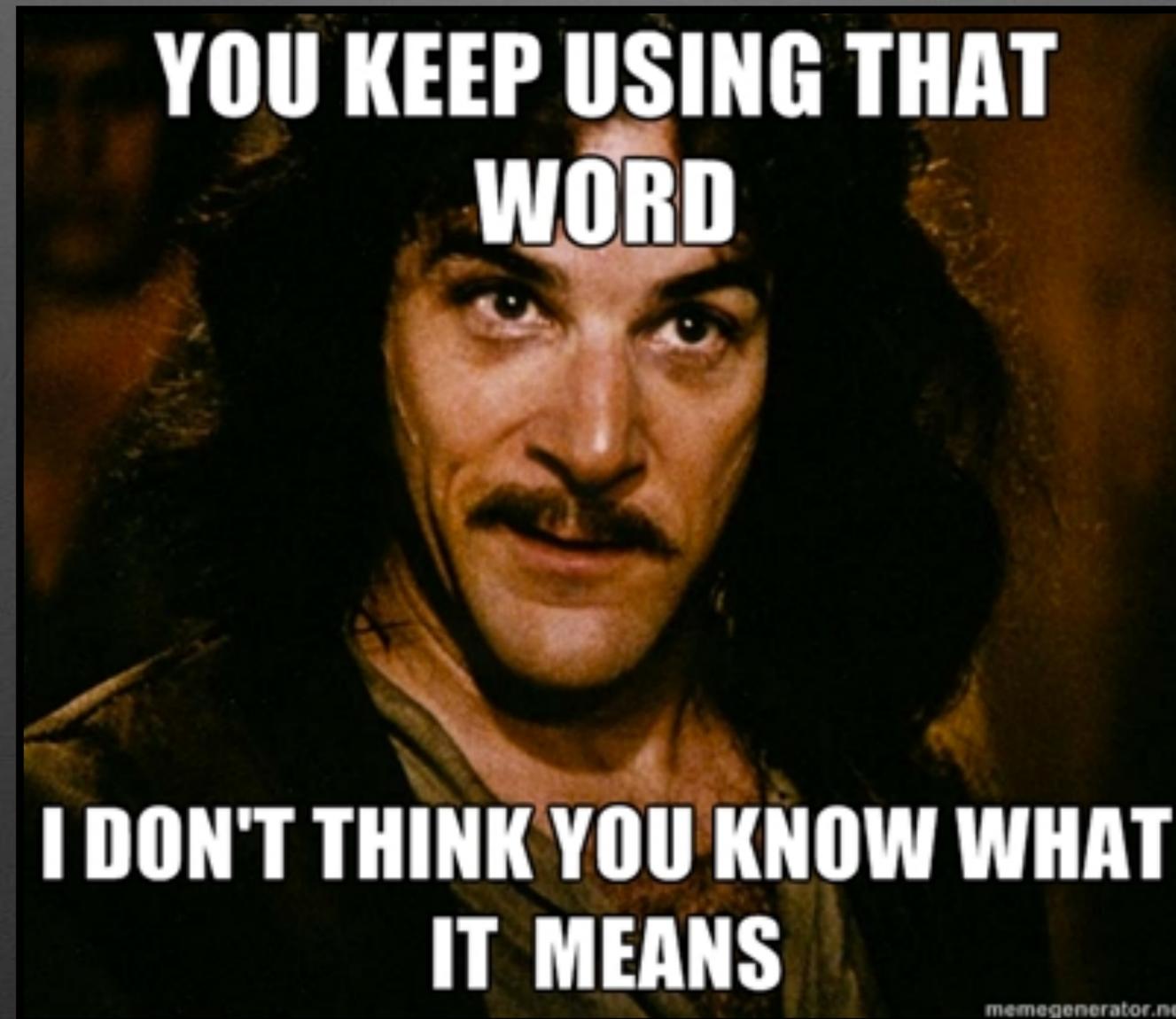


it's probably fine.

(it might be fine?)

Your system is never entirely 'up'

Many catastrophic states exist at any given time.





Distributed systems have an infinitely long list of almost-impossible failure scenarios that make staging environments particularly worthless.

this is a black hole for engineering time

TESTING IN PRODUCTION

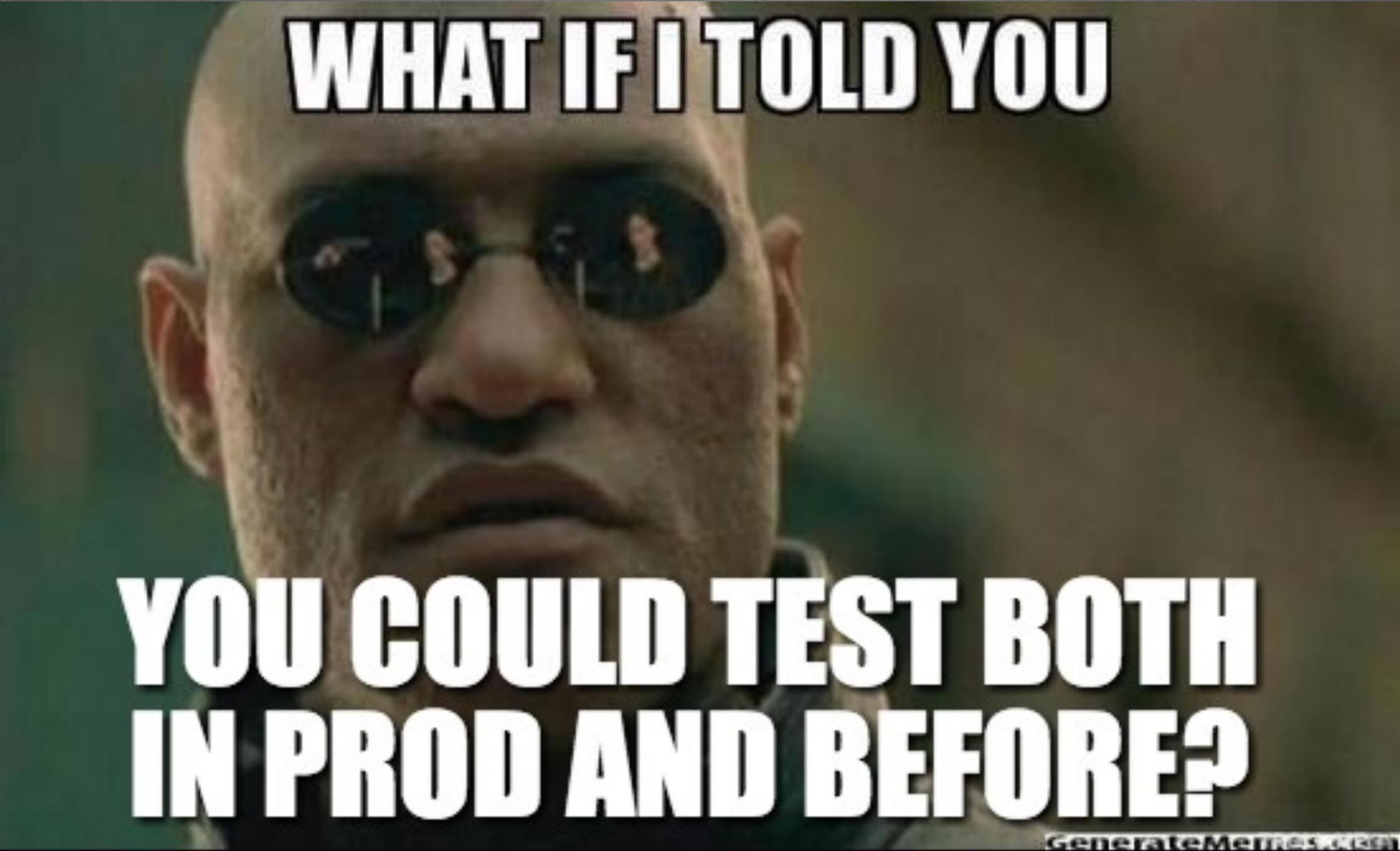


TELL ME MORE

memegenerator.net

You do it.
You have to do it.
Do it well.

And don't do it blind.

A close-up image of Morpheus from the movie The Matrix, wearing his signature sunglasses. The image is used as a background for a meme. The text is overlaid in white, bold, sans-serif font with a black outline. The top text reads "WHAT IF I TOLD YOU" and the bottom text reads "YOU COULD TEST BOTH IN PROD AND BEFORE?".

WHAT IF I TOLD YOU

**YOU COULD TEST BOTH
IN PROD AND BEFORE?**



Monitoring

Observability



testing in production
software engineers on call/ownership
fewer paging alerts
feels like the missing link for literally everything

Let's try some examples!

LAMP stack vs distributed system



“Photos are loading slowly for some people. Why?”

(LAMP stack)

The app tier capacity is exceeded. Maybe we rolled out a build with a perf regression, or maybe some app instances are down.

monitor these things

Errors or latency are high. We will look at several dashboards that reflect common root causes, and one of them will show us why.

DB queries are slower than normal. Maybe we deployed a bad new query, or there is lock contention.

Monitoring

Characteristics

(LAMP stack)



- Known-unknowns predominate
- Intuition-friendly
- Dashboards are valuable.
- Monolithic app, single data source.
- The health of the system more or less accurately represents the experience of the individual users.

Monitoring

Best Practices



- Lots of actionable active checks and alerts
- Proactively notify engineers of failures and warnings
- Maintain a runbook for stable production systems
- Rely on clusters and clumps of tightly coupled systems all breaking at once

Monitoring

“Photos are loading slowly for some people. Why?”

(microservices)

Any microservices running on c2.4xlarge instances and PIOPS storage in us-east-1b has a 1/20 chance of running on degraded hardware, and will take 20x longer to complete for requests that hit the disk with a blocking call. **This disproportionately impacts people looking at older archives due to our fanout model.**

wtf do i ‘monitor’ for?!

Canadian users who are using the French language pack on the iPad running iOS 9, are hitting a firmware condition which makes it fail saving to local cache ... **which is why it FEELS like photos are loading slowly**

Our newest SDK makes db queries sequentially if the developer has enabled an optional feature flag. **Working as intended; the reporters all had debug mode enabled. But flag should be renamed for clarity sake.**

Monitoring?!?

Problems Symptoms

(microservices)

"I have twenty microservices and a sharded db and three other data stores across three regions, and everything seems to be getting a little bit slower over the past two weeks but nothing has changed that we know of, and oddly, latency is usually back to the historical norm on Tuesdays.



"All twenty app micro services have 10% of available nodes enter a simultaneous crash loop cycle, about five times a day, at unpredictable intervals. They have nothing in common afaik and it doesn't seem to impact the stateful services. It clears up before we can debug it, every time."

"Our users can compose their own queries that we execute server-side, and we don't surface it to them when they are accidentally doing full table scans or even multiple full table scans, so they blame us."

Observability

Still More Symptoms

(microservices)

“Several users in Romania and Eastern Europe are complaining that all push notifications have been down for them ... for days.”

“Sometimes a bot takes off, or an app is featured on the iTunes store, and it takes us a long long time to track down which app or user is generating disproportionate pressure on shared components of our system (esp databases). It’s different every time.”

“Disney is complaining that once in a while, but not always, they don’t see the photo they expected to see — they see someone else’s photo! When they refresh, it’s fixed. Actually, we’ve had a few other people report this too, we just didn’t believe them.”

“We run a platform, and it’s hard to programmatically distinguish between problems that users are inflicting themselves and problems in our own code, since they all manifest as the same errors or timeouts.”

Observability

These are all unknown-unknowns
that may have never happened before, or ever happen again

(They are also the overwhelming majority of what you have
to care about for the rest of your life.)



Characteristics

(microservices/complex systems)

- Unknown-unknowns are most of the problems
- “Many” components and storage systems
- You cannot model the entire system in your head. Dashboards may be actively misleading.
- The hardest problem is often identifying which component(s) to debug or trace.
- The health of the system is irrelevant. The health of each individual request is of supreme consequence.

Observability

Known-unknowns

- A support problem
- Predictable time scale
- Use a fucking dashboard, then automate it out of existence



Unknown-unknowns

- An engineering problem
- Open-ended time scale
- Require creativity

Very, very, very different tooling.

Observability for SWEs and the Future™



well-instrumented
high cardinality
high dimensionality
event-driven
structured
well-owned
sampled
tested in prod.



You're not bad at debugging,
your tools were designed for different problems

“What problems? How are they different?”



Hunting unknowns:

Instrumentation

Events, not metrics

Few (if any) dashboards

Sampling, not preaggregation

Test in production

Fewer alerts



Instrumentation



Get inside the software's head
Explain it back to yourself/a naive user
The right level of abstraction is key
Wrap all network calls, etc
Open up all black boxes to inspection

Events, not metrics



Cardinality

Context

Structured data

Tracing+events

*(trick question.. you'll need both
but you'll rely on events more and more)*

Metrics

(cardinality)

*Some of these ...
might be ...
useful ...
YA THINK??!*

High cardinality will save your ass.

UUIDs
db raw queries
normalized queries
comments
firstname, lastname
PID/PPID
app ID
device ID
HTTP header type
build ID
IP:port
shopping cart ID
userid
... etc

High cardinality is not a nice-to-have

You must be able to break down by 1/millions and
THEN by anything/everything else

'Platform problems' are now everybody's problems



Looking for a needle in your haystack? Be descriptive, add unique identifiers.

Read-time aggregation lets you compute percentiles, derived columns.

```
[{"Timestamp": "Sat Sep  2 00:30:56 UTC 2017"  
  "api_version": "1"  
  "availability_zone": "us-east-1b"  
  "batch": false  
  "build_id": "4715"  
  "chosen_partition": 31  
  "content_length": 156785  
  "dataset_columns": 35  
  "dataset_expand_json_depth": 0  
  "dataset_id": 3915  
  "dataset_name": "kubernetes-resource-metrics"  
  "dataset_partitions": "[3,31,35]"  
  "dynsample_key": "batch,3915,784,202"  
  "dynsample_rate": 1  
  "env": "production"  
  "event_columns": 23  
  "event_time": "0001-01-01T00:00:00Z"  
  "extra_headers": "Accept-Encoding: gzip, Connection: keep-alive, Content-Length: 156785, Content-Type: application/json, X-Forwarded-Port: 443, X-Forwarded-Proto: https"  
  "instance_type": "c4.large"  
  "json_decoding_duration_ms": 1.172  
  "memory_inuse": 1.172  
  "method": "POST"  
  "nested_json": false  
  "nested_json_depth": 0  
  "num_goroutines": 270  
  "oversize_len_longest": 0  
  "oversize_num_columns": 0  
  "oversize_total_bytes": 0  
  "prep_partition_info_duration_ms": 0.004373  
  "process_uptime_seconds": 2545  
  "remote_addr": "10.0.54.83:2158"  
  "request_id": "ab060d1f-84bf-4180-aa2a-84cdf0c0008a"
```

Metrics
(structured data)



Events tell stories.

Arbitrarily wide events mean you can amass more and more context over time. Use sampling to control costs and bandwidth.

Structure your data at the source to reap massive efficiencies over strings.



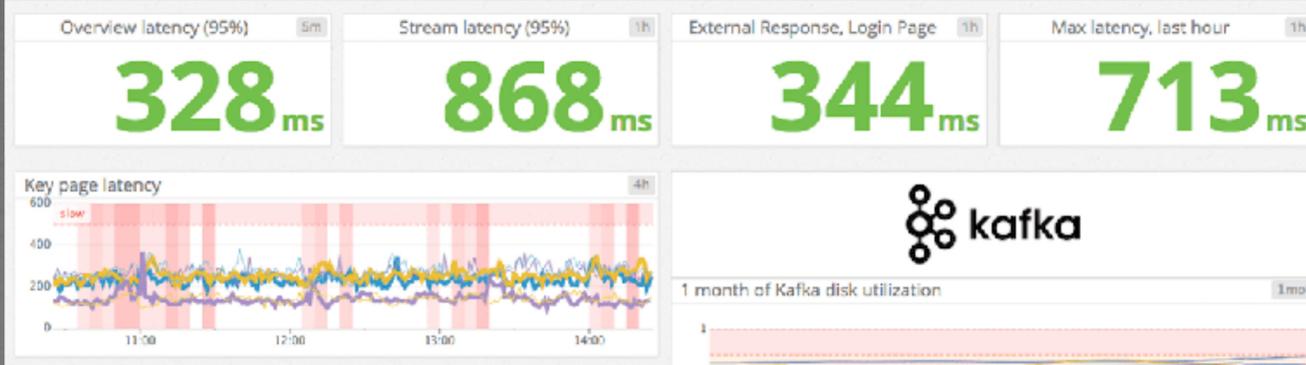
Context

("Logs" are just a transport mechanism for events)

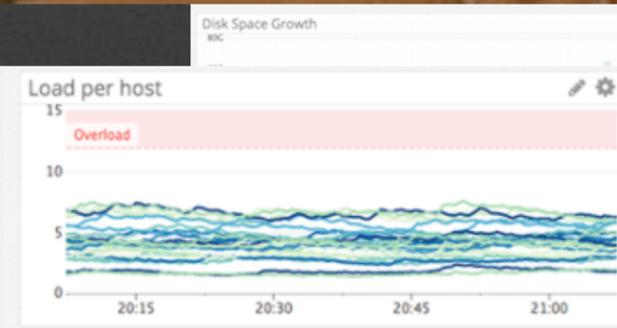
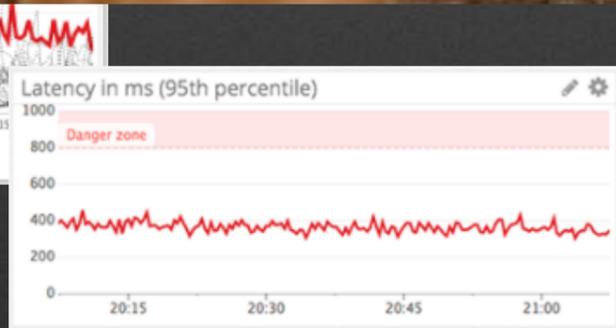
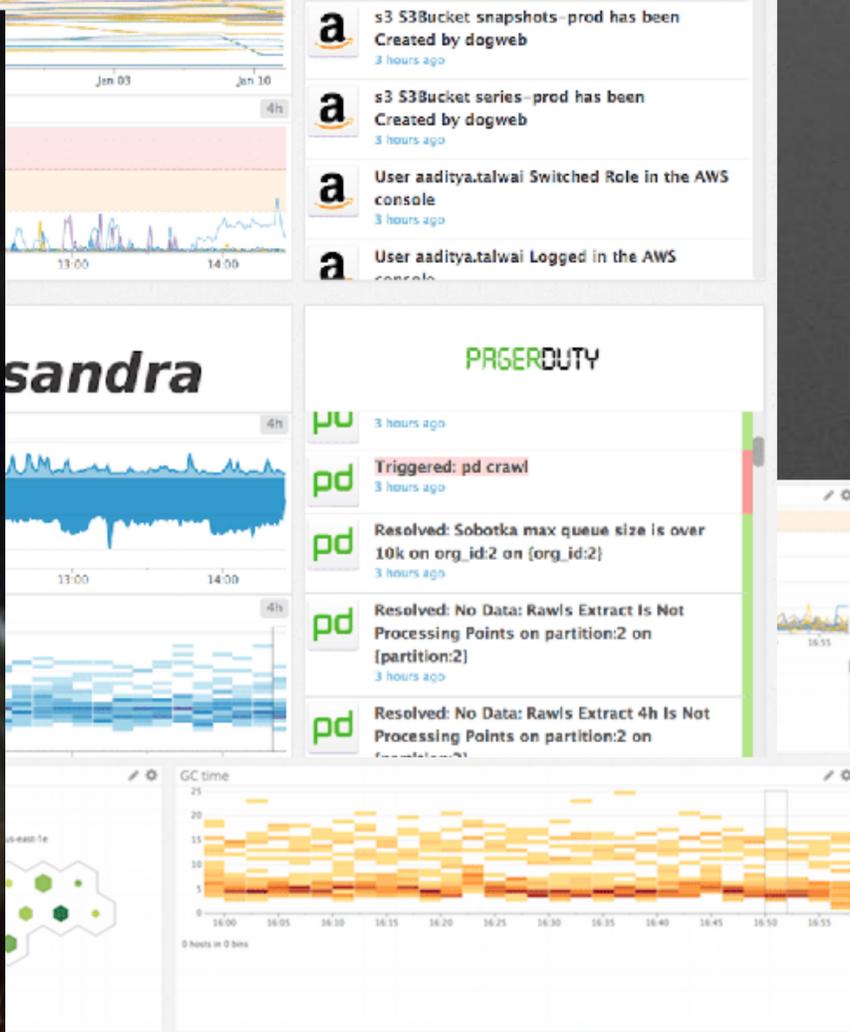
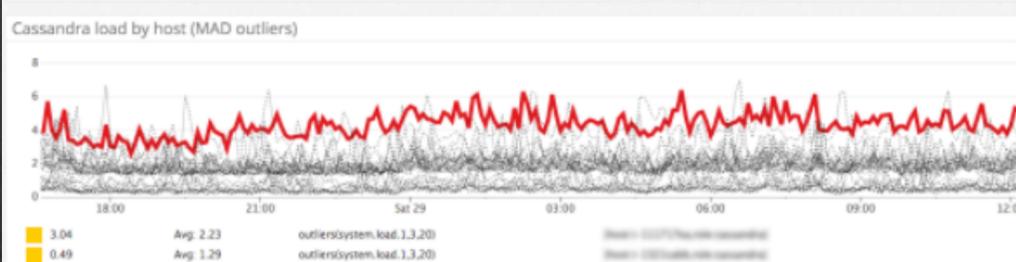
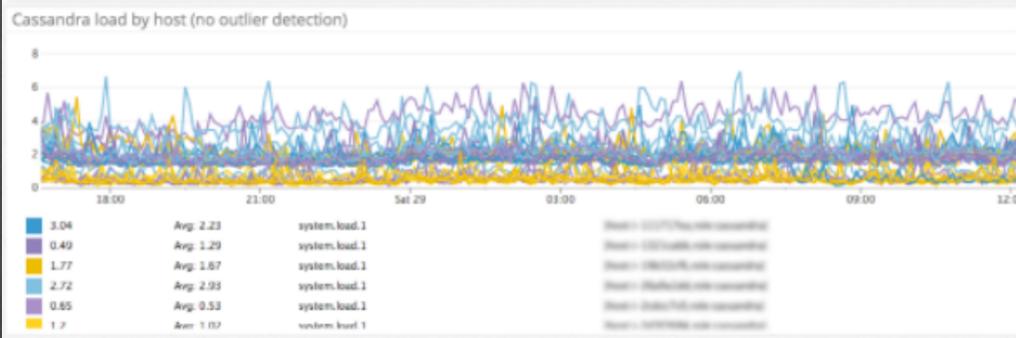


Dashboards??





- AWS CloudTrail
- User olivier.vielpeau Logged in the AWS console 2 hours ago
 - s3 S3Bucket series-prod has been Created by dogweb 3 hours ago
 - s3 S3Bucket series-prod has been Created by dogweb 3 hours ago
 - s3 S3Bucket snapshots-prod has been Created by dogweb 3 hours ago
 - s3 S3Bucket series-prod has been Created by dogweb 3 hours ago
 - User aaditya.talwai Switched Role in the AWS console 3 hours ago
 - User aaditya.talwai Logged in the AWS console 3 hours ago



Dashboards

Dashboards

Artifacts of past failures.

Jumps to an answer, instead of starting with a question

You don't know what you don't know.



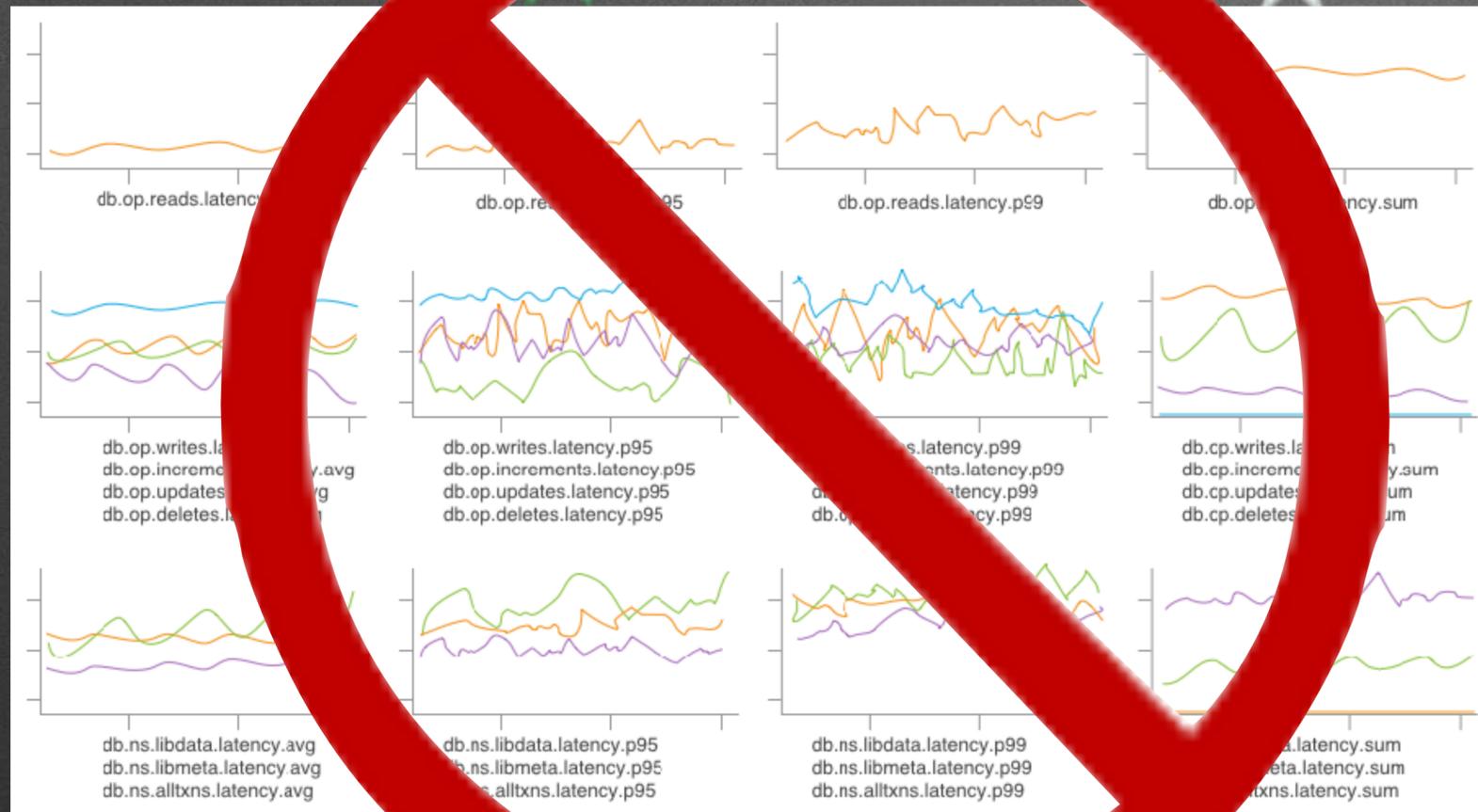
Exploratory

Interactive

Iterative

Fast

Raw



Dashboards must reward interaction



**Unknown-unknowns
demand explorability
and an open mind.**

Raw requests:

sampling, not aggregation



Aggregation is a one-way trip

Destroying raw events eliminates your ability to ask new questions.
Forever.

PLEASE
DON'T SMOOSH AWAY ALL
MY PRECIOUS DETAIL!!!



Aggregates are the devil

Aggregates destroy your precious details.

You need MORE detail and MORE context.



Aggregates

You MUST be able to ask arbitrary new questions about your events on the fly. Aggregating at write time destroys your ability to do this.

Forever.

Aggregates

You can't hunt needles if your tools don't handle extreme outliers, aggregation by arbitrary values in a high-cardinality dimension, super-wide rich context...

Black swans are the norm

you must care about max/min, 99%, 99.9th, 99.99th, 99.999th ...



Row Requests

“Sum up all the time spent holding the user.* table lock by INSERT queries, broken down by user id and the size of the object written, and show me any users using more than 30% of the overall row lock.”

“Latency seems elevated for HTTP requests. Requests can loop recursively back into the API multiple times; are requests getting progressively slower as the iteration stack gets deeper? What is the MAX recursive call depth, and max latency over the past day? Is it still growing? What do the 100 slowest have in common?”

“Show me all the 50x errors broken down by user id or app id. Show me all the abandoned carts with the most items in them. Show me the users rate limited in the past hour, broken down by browser type or mobile device type and release version string.”

Raw data examples

Zero users care what the “system” health is

All users care about THEIR experience.

Nines don't matter if users aren't happy.

Raw Requests matter if users aren't happy.

Nines don't matter if users aren't happy.

Services need owners, not operators.

Observability:

must be designed for generalist SWEs.

SaaS, APIs, SDKs.

Ops lives on the other side of an API



Engineers



Monitoring **Instrumentation** is part of building software

Engineers

Software engineers spend too much time looking at code in elaborately falsified environments, and not enough time **observing it in the real world.**

Tighten feedback loops. Give developers the observability tooling they need to become fluent in production and to debug their own systems.

*We aren't "writing code".
We are "building systems".*

ops:monitoring::SWE:tests

known unknowns -> unknown unknowns

~~Test-Driven Development~~

Observability-Driven Development

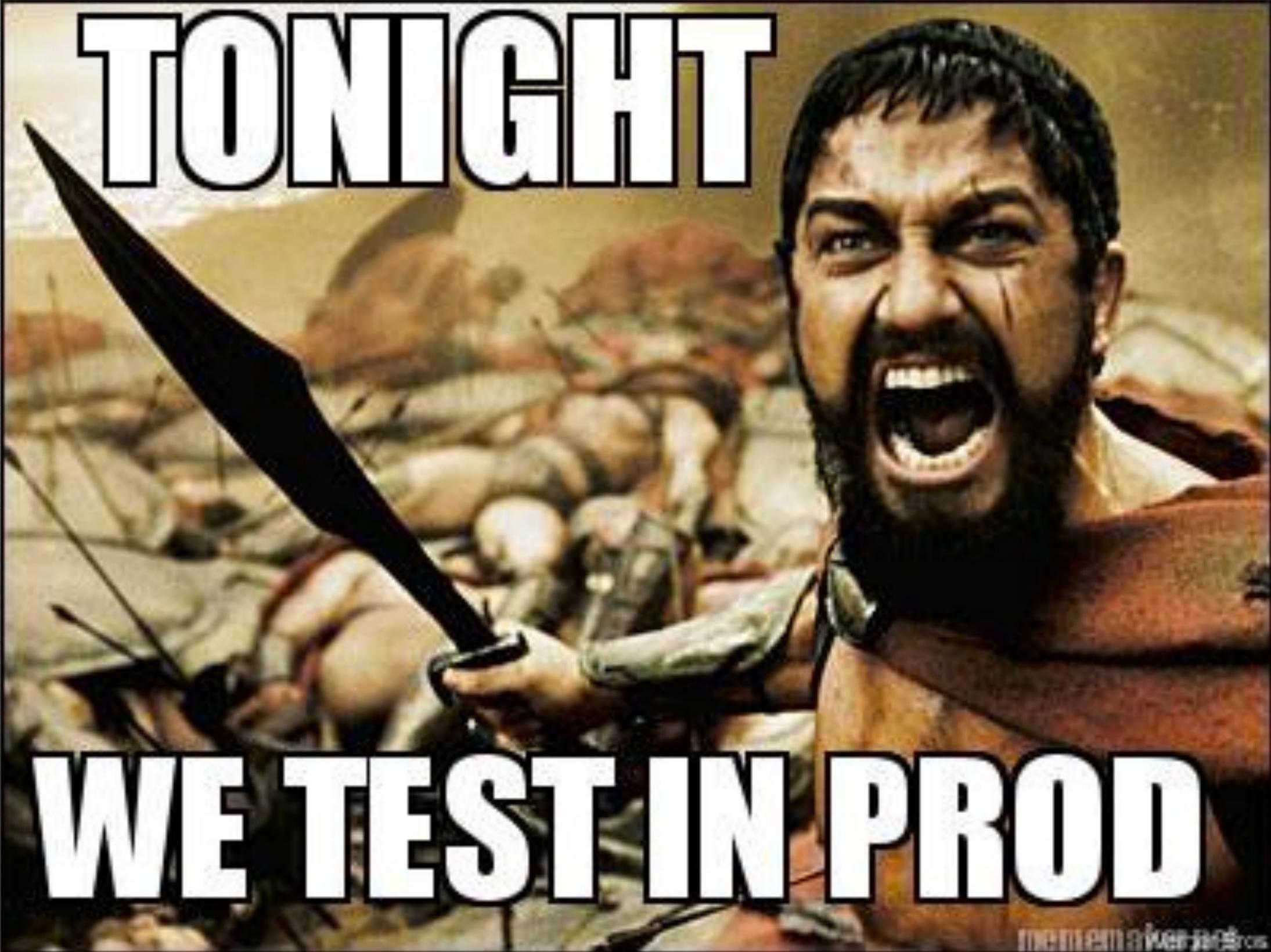


Watch it run in production.

Accept no substitute.

Get used to observing your systems when they AREN'T on fire.

Engineers



TONIGHT

WE TEST IN PROD

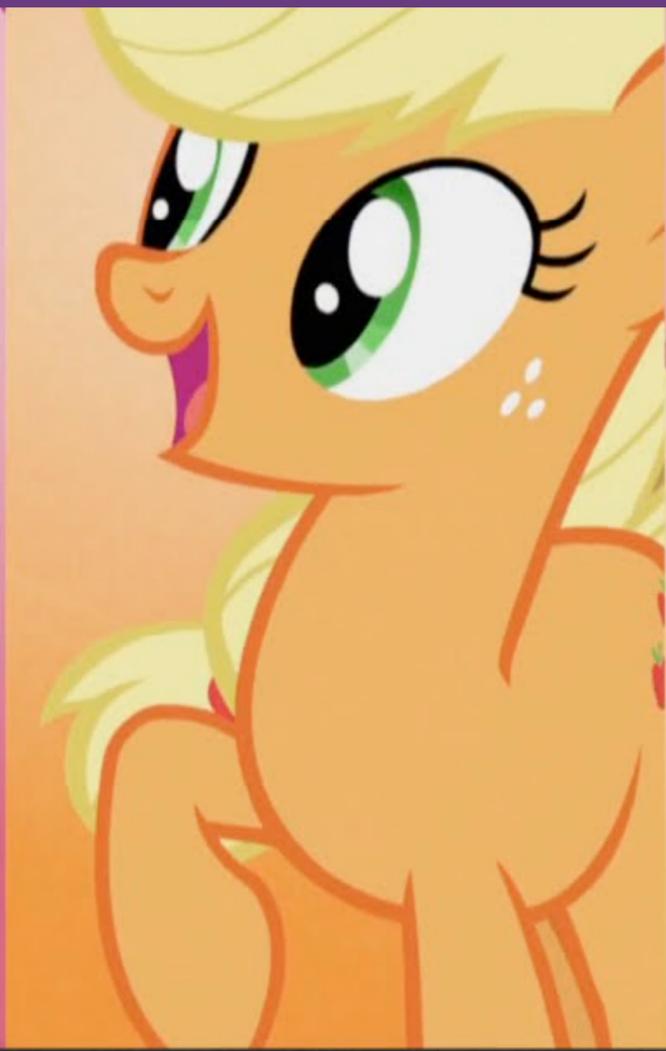
Observability for the Future™



well-instrumented
high cardinality
high dimensionality
event-driven
structured
well-owned
sampled
tested in prod.



**You win ...
Drastically fewer paging alerts!**





Charity Majors
@mipsytipsy

