Production Hardened Services

Nathaniel T. Schutta
@ntschutta

Ah services!

Service all the things!

Seems like everything is aaS these days…

Infrastructure. Container. Platform.
Software. Function. Pizza.

Pretty sure about that last one…

Architecture as a Service…

How many services do *you* have in your world these days?

Turns out, microservices aren't a panacea!

It can be a real challenge to maintain a healthy micro(services)biome.

Should we just go back to monoliths we release semiannually?

Is it too late to change careers?

How did we get here?

Nothing new really.

Remember when SOA was the new hotness?

Everything had to be all SOA all the time!

There were books and talks and blogs (remember those?), it was great!

How about API first?

Popularized in certain quarters.

sitorsquat.com

Powered by Charmin enjoy the go

Gotta Go When You're on the Go?

Now you can take sit or squat wherever you go!
if you already have the app, be sure to update it to the new version on your phone.

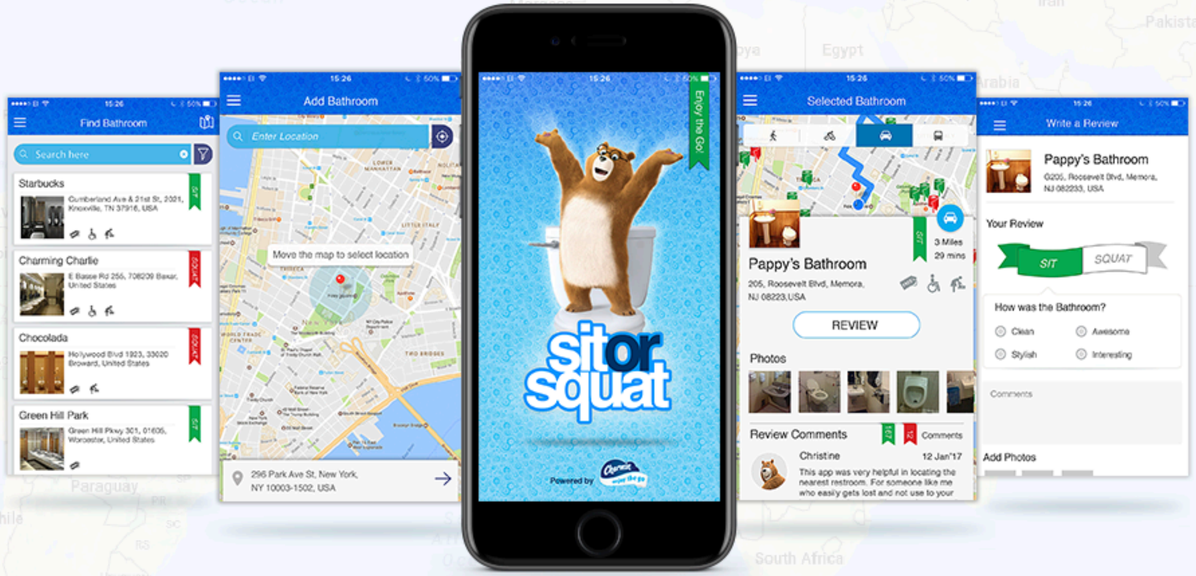
We have moved to an app-only version to give you a better experience when you gotta go. We hope you enjoy our new easy-to-go mobile app!

Download our improved SitOrSquat app that is available for FREE on: App Store | Google Play

Updates include new security upgrades and improved map functionality!
Check out more information on the SitOrSquat app by clicking here.

Find Charmin online! Charmin.com

Enjoy our Charmin Pooptastic iMessage stickers by clicking here.



Helpful?

How about this one?

darksky.net

Dark Sky API

DOCS FAQ LOG IN SIGN UP

Dark Sky API

The easiest, most advanced, weather API on the web.

TRY FOR FREE

Easy to Use

- Get setup in minutes
- Developer friendly
- Clear and concise documentation
- Pay-as-you-go pricing

Weather Conditions

- Forecasts and current conditions
- Global coverage
- Historical data
- Severe weather alerts

Advanced Data

- Cutting-edge forecast modeling techniques
- Minute-by-minute hyperlocal forecasts
- Backed by a wide range of data sources

LEARN MORE

Bet you use that one everyday.
Maybe without knowing it.

One of these perhaps?

developers.google.com

Google

Search for services, methods, and recent requests... Loading...

APIs Explorer

- Services
- All Versions
- Request History

Accelerated Mobile Pages (AMP) URL API	v1	This API contains a single method, batchGet. Call this method to retrieve the AMP URL (and equivalent AMP Cache URL) for given public URL(s).
Ad Exchange Buyer API	v1.4	Accesses your bidding-account information, submits creatives for validation, finds available direct deals, and retrieves performance reports.
Ad Exchange Buyer API II	v2beta1	Accesses the latest features for managing Ad Exchange accounts, Real-Time Bidding configurations and auction metrics, and Marketplace programmatic deals.
Ad Exchange Seller API	v2.0	Accesses the inventory of Ad Exchange seller users and generates reports.
Admin Reports API	reports_v1	Fetches reports for the administrators of G Suite customers about the usage, collaboration, security, and risk for their users.
AdSense Host API	v4.1	Limited Availability Generates performance reports, generates ad codes, and provides publisher management capabilities for AdSense Hosts.
AdSense Management API	v1.4	Accesses AdSense publishers' inventory and generates performance reports.
Android Device Provisioning Partner API	v1	Automates reseller integration into zero-touch enrollment by assigning devices to customers and creating device reports.
Android Management API	v1	The Android Management API provides remote enterprise management of Android devices and apps.
APIs Discovery Service	v1	Provides information about other Google APIs, such as what APIs are available, the resource, and method details for each API.
BigQuery API	v2	A data platform for customers to create, manage, share and query data.
BigQuery Data Transfer API	v1	Transfers data from partner SaaS applications to Google BigQuery on a scheduled, managed basis.
Blogger API	v3	Limited Availability API for access to the data within Blogger.
Books API	v1	Searches for books and manages your Google Books library.
Calendar API	v3	Manipulates events and other calendar data.
Cloud Monitoring API	v2beta2	Accesses Google Cloud Monitoring data.
Cloud Source Repositories API	v1	Access source code repositories hosted by Google.
Cloud Spanner API	v1	Cloud Spanner is a managed, mission-critical, globally consistent and scalable relational database service.
Cloud SQL Administration API	v1beta4	Creates and configures Cloud SQL instances, which provide fully-managed MySQL databases.
Cloud Storage JSON API	v1	Stores and retrieves potentially large, immutable data objects.
Cloud Tasks API	v2beta2	Manages the execution of large numbers of distributed requests. Cloud Tasks is in Alpha.
Cloud Tool Results API	v1beta3	Reads and publishes results from Firebase Test Lab.
Cloud User Accounts API	vm_alpha	Creates and manages users and groups for accessing Google Compute Engine virtual machines.

What caused this Cambrian explosion of APIs?

Technology changes.

Apple Inc.

iPhone X

Overview iOS Tech Specs Models and Pricing

Proliferation of computers in everyone’s pockets.

Commoditized hardware.

The Cloud!

0
11
VOLUME

0
11
VOLUME

Companies were altering their approach too.

Little company called Amazon made a major change.

Well, it had happened years earlier.

But a publicly shared rant detailed it.

https://plus.google.com/+RipRowan/posts/eVeouesvaVX

Steve Yegge - the Bezos mandate.

All data will be exposed through a public service interface.

These interfaces are *the*
communication method between teams.

No other form of communication is allowed. No direct reads, links etc.

No back doors.

All services must be designed to be public. No exceptions.

Don’t want to do this? You’re fired.

Unsurprisingly, things began to change.

And we learned some things.

Calls bounce between 20 services…
where was the problem?

Who do we page?

How do we monitor this web of services?

How do I even *find*
these other services?

Debugging gets harder…

We’ve seen this story continue today.

Can’t swing a dry erase marker without hitting someone talking about…

Microservices!

Bounded Context.

Domain-Driven Design.

Arguments over the definition of microservice…

Rewrite it in two weeks.

Miniservice. Picoservice.

What do we even mean by "application" today?!?

How about functions then?

ENGINEERING

Introducing Spring Cloud Function

ENGINEERING MARK FISHER JULY 05, 2017 8 COMMENTS

Spring Cloud Function is a new project with the following high-level goals:

- Promote the implementation of business logic via functions.
- Decouple the development lifecycle of business logic from any specific runtime target so that the same code can run as a web endpoint, a stream processor, or a task.
- Support a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
- Enable Spring Boot features (auto-configuration, dependency injection, metrics) on serverless providers.

Just as Spring has always promoted a plain old java object (POJO) based programming model, Spring Cloud Function promotes a programming model based on plain old functions. By that we mean the core interfaces defined in the `java.util.function` package: `Function`, `Consumer`, and `Supplier`.

Implementations of those types can be registered as beans either explicitly or implicitly via classpath scanning enabled by `@FunctionScan`. The parameter and/or return types can optionally use Reactor's `Flux`, which is a Reactive Streams `Publisher`. That enables interoperability with other Reactive Streams components, even those that are based on other implementations, such as RxJava 2, and it brings reactive features such as non-blocking IO and back-pressure to this processing model (for more information see Project Reactor). Whenever the parameter and/or return types are not `Flux`, Spring Cloud Function wraps them so that functions can interoperate via `Flux`. For simple item-at-a-time processing use-cases, you can keep it simple:

```
public class Greeter implements Function<String, String> {
  public String apply(String name) {
    return "Hello " + name;
  }
}
```

But if you need to implement functions that handle a dataset as the unit of processing, via windowing or reduce operations, you can use `Flux` types:

Have we found a golden hammer yet?

Bet that would be helpful during your next retro!

Turns out there are still engineering issues we have to overcome.

It isn’t all puppies and rainbows.

Sorry.

Turns out, those things Yegge mentions…are still things.

Culture.

Culture? What does that have to do with my services?

Culture is hard.

Culture is where good ideas go to die.

Culture is really hard to change.

What did Bezos resort to?

You probably don’t have that much clout in your organization.

Microservices offer many advantages.

Scalability.

Developer productivity.

Polyglot platforms.

Still no silver bullets…

When we find ourselves presented with technology that promises to offer us drastic improvements, we need to look for the trade-offs.

— Susan J. Fowler
Production-Ready Microservices

Chances are, our org structure isn't helping matters much.

LINKS

Text of Paper

Wikipedia: Fred Brooks

Wikipedia: Conway's Law

In 1967 I submitted a paper called "How Do Committees Invent?" to the *Harvard Business Review*. *HBR* rejected it on the grounds that I had not proved my thesis. I then submitted it to *Datamation*, the major IT magazine at that time, which published it April 1968. The text of the paper is here.

Here is one form of the paper's thesis:

> **Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.**

Fred Brooks cited the paper and the idea in his elegant classic "The Mythical Man-Month," calling it "Conway's Law." The name stuck.

Following is an extract from an article in Wikipedia. (The concept originated in the software world but is not limited to any specific domain.)

> Conway's law was not intended as a joke or a Zen koan, but as a valid sociological observation. It is a consequence of the fact that two software modules A and B cannot interface correctly with each other unless the designer and implementer of A communicates with the designer and implementer of B. Thus the interface structure of a software system *necessarily* will show a congruence with the social structure of the organization that produced it.

Brooks recognized that the law has important corollaries in management theory. Here is one stated in the paper.

> Because the design that occurs first is almost never the best possible, the prevailing system concept may need to change. Therefore, flexibility of organization is important to effective design.

In retrospect, *HBR*'s basis for rejecting the paper says more about differences in notions of "proof" than it does about the paper.

[Note: I assume no responsibility for information in other Web sites. The reference to Fred Brooks in Wikipedia, for example, was accurate to the best of my knowledge at the time I created the link to it, but it is subject to change beyond my control (as is all information on the Web not in this site).]

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

— Melvin Conway

How do you create a series of small isolated services…

If your organization isn’t a set of small, isolated teams?

Inverse Conway's Law.

Evolve your teams towards the architectural end state you desire.

https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

Leads to some interesting outcomes.

Teams are often siloed. But microservices have to work together.

How do we get those teams to work together effectively?

My incentives may not align with yours…how do we solve for that?

How do we build out infrastructure for many disparate teams?

How do we staff up the operations team?

Application to Operations cannot be 1-1!

You build it, you own it.

Are your teams ready for pager duty?

Can’t just pitch it over the wall to Ops anymore.

Technical sprawl.

Polyglot!

It’s great right? Each team can use just the right tool for the job!

Every developer, every team will have their favorite tools, languages, etc.

Teams will have their pipeline preferences, meaningful metrics…

Leads to an awful lot of ways to do a given thing.

How do we staff up? Go, Haskell, Java, .NET, C++, Ruby, Python, JavaScript?

How many libraries will we need to support all of that?

Can we stay current?

heartbleed.com

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

What leaks in practice?

We have tested some of our own services from attacker's perspective. We attacked ourselves from outside, without leaving a trace. Without using any privileged information or credentials we were able steal from ourselves the secret keys used for our X.509 certificates, user names and passwords, instant messages, emails and business critical documents and communication.

How to stop the leak?

As long as the vulnerable version of OpenSSL is in use it can be abused. Fixed OpenSSL has been released and now it has to be deployed. Operating system vendors and distribution, appliance vendors, independent software vendors have to adopt the fix and notify their users. Service providers and users have to install the fix as it becomes available for the operating systems, networked appliances and software they use.

BUSINESS
SEP 14 2017, 3:21 PM ET

Equifax Hackers Exploited Months-Old Flaw

by BEN POPKEN

SHARE
Share
Tweet
Email
Print

Equifax announced late Wednesday that the source of the hole in its defenses that enabled hackers to plunder its databases was a massive server bug first revealed in March.

For the rest of the IT world, fixing that flaw was a "hair on fire moment," a security expert said, as companies raced to install patches and secure their servers. But at Equifax, criminals were able to pilfer data from mid-May to July, when the credit bureau says it finally stopped the intrusion.

Equifax, Software Company Blame Each Other for Security Breach 1:52

"We know that criminals exploited a U.S. website application vulnerability," Equifax said in an update on its website Wednesday night. "The vulnerability was Apache Struts CVE-2017-5638." Equifax said it was working with a leading cybersecurity firm, reported to be Mandiant, to investigate the breach. Mandiant declined an NBC News request for comment.

Related: The One Move to Make After Equifax Breach

The Apache Software Foundation, which oversees the Apache Struts project, said in a press release Thursday that a software update to patch the flaw was

advertisement

FROM THE WEB
Sponsored Links

Find Out In One Minute If You Pre-Qualify For A Citi Card
Citi

Why These 10 SUVs are the Cream of the Crop
Kelley Blue Book

by Taboola

MORE FROM NBC NEWS

It cannot be a free for all.

You will need some guardrails.

“Use any language as long as it runs on the JVM.”

Pick from these 4 flavors. Won’t work for you? Let’s talk.

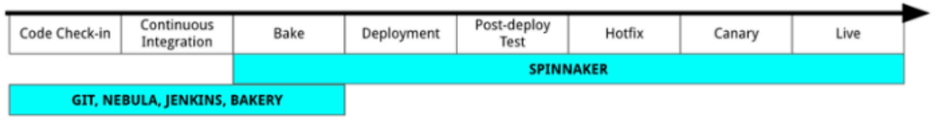
Focus on “paved roads.”

A Medium Corporation

Netflix Technology Blog Follow
Learn more about how Netflix designs, builds, and operates our systems and engineering organizations
Mar 9, 2016 · 8 min read

How We Build Code at Netflix

How does Netflix build code before it's deployed to the cloud? While pieces of this story have been told in the past, we decided it was time we shared more details. In this post, we describe the tools and techniques used to go from source code to a deployed service serving movies and TV shows to more than 75 million global Netflix members.



The above diagram expands on a previous post announcing Spinnaker, our global continuous delivery platform. There are a number of steps that need to happen before a line of code makes it way into Spinnaker:

- Code is built and tested locally using Nebula
- Changes are committed to a central git repository
- A Jenkins job executes Nebula, which builds, tests, and packages the application for deployment
- Builds are "baked" into Amazon Machine Images
- Spinnaker pipelines are used to deploy and promote the code change

196 4
Next story
Deploying Features Under Cove...

Here is a well worn path, we know it works, we support it.

MINIMUM
MAINTENANCE
ROAD
TRAVEL AT YOUR OWN RISK

You build it, you own it.

Sprawl tends to exacerbate our accumulation of technical debt.

Must be vigilant about paying that down over time.

Multiple failure modes.

Microservice ecosystems are far more complex than monoliths.

Small pieces changing all the time.

How do you reason about your system now?

No one could have predicted this.

Our systems will fail in all sorts of new unique ways!

NETFLIX

SIMIAN ARMY

https://github.com/Netflix/SimianArmy

Chaos Engineering.

https://medium.com/production-ready/chaos-monkey-for-fun-and-profit-87e2f343db31

Hate to break it to you…
your systems will fail.

We cannot prevent it but we can certainly prepare for it.

Resources do not scale to infinity.

There will be competition between teams for hardware, staff, priorities.

Do not underestimate the battle for headcount.

Every manager could use just one more engineer. Or ten.

Every VP thinks their portfolio should get the lions share of the budget.

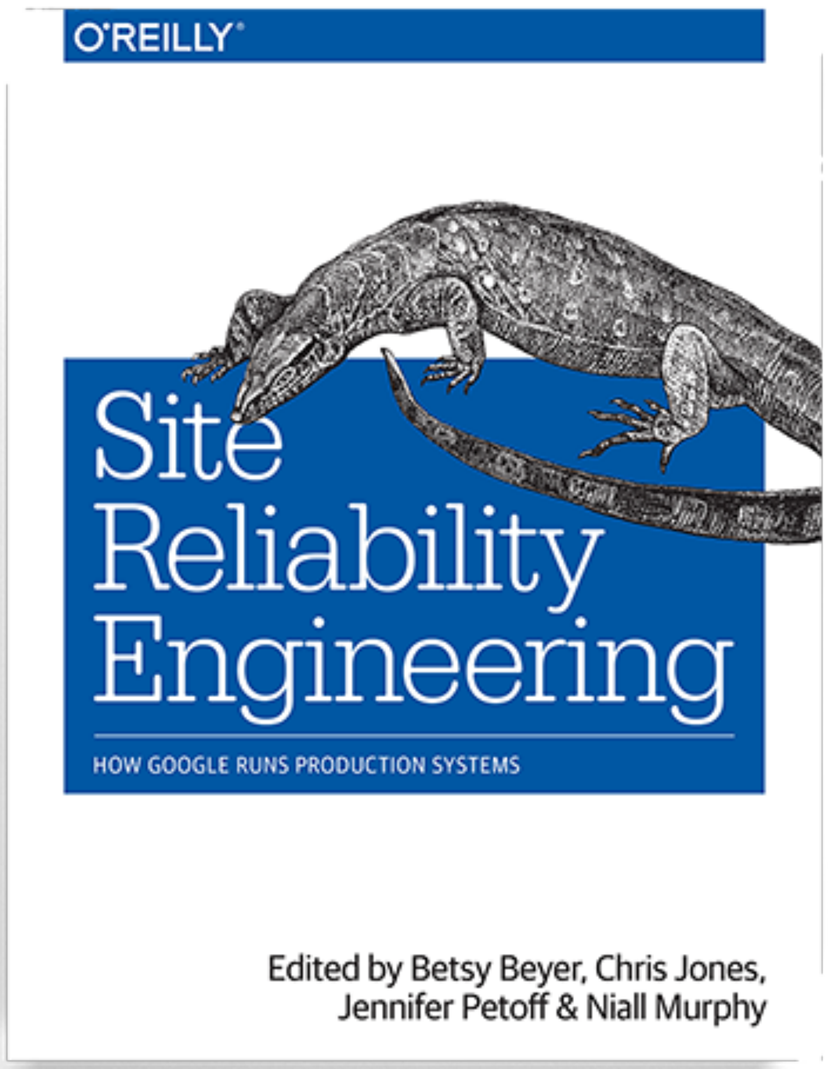
No one could have predicted this.

All services are equal. Some services are more equal than others.

Defining our SLOs is a critical step towards production hardened services.

Availability is one of our most important objectives.

What is our availability goal?



https://landing.google.com/sre/book.html

SLAs vs. SLOs.

Terminology is often misused.

SLI - service level indicator.

A measure of some aspect of our system.

Latency, error rate, throughput…

Availability.

Percentage of time your service is available.

How much downtime can you tolerate?

99%: 7.20 hours a month,
14.4 minutes a day.

99.9%: 8.76 hours PER YEAR.
1.44 minutes a day.

99.99%: 4.38 minutes A MONTH.
8.66 seconds a day.

99.999%: 6.05 seconds A WEEK,
864.3 milliseconds a day.

By the way, the GKE's availability SLA is 99.5%. Just saying.

https://cloud.google.com/kubernetes-engine/sla

SLO - service level objective.

Our target value or a range of values.

Our SLO for availability is 99.9%.

Our average response time should be less than 100 milliseconds.

Can be very tricky to pick an SLO!
More on that in a minute.

We don’t always get to choose our SLOs.

Our users may have a thing or two to say about what they want!

We are also subject to the things we depend on.

If you depend on a service that only guarantees 99.9% availability…

You cannot guarantee 99.99%!

SLA - service level agreement.

Assign a consequence to missing/meeting an SLO.

Often contractual and involve some kind of financial penalty.

If there is no consequence, we're talking about an SLO.

SLAs are the realm of product and legal decisions.

Many people say SLA
when they mean SLO.

Pedantic much?

Ubiquitous language and all that…

Everyone wants 99.999%.

Everyone wants hot/hot.

Until they see the price tag.

If you have to ask…

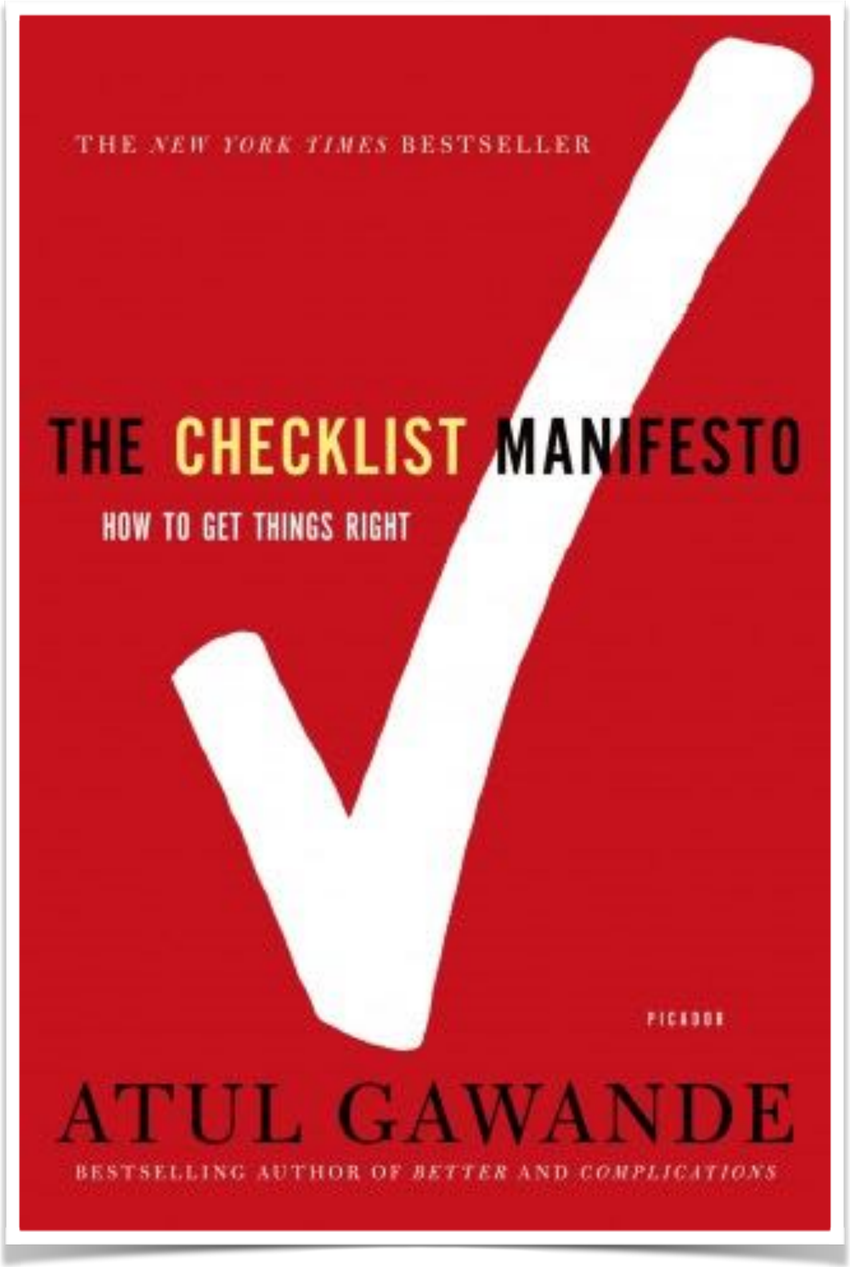
When we refer to an application or microservice as "production-ready," we confer a great deal of trust upon it: we trust it to behave reasonably, we trust it to perform reliably, we trust it to get the job done…

— Susan J. Fowler
Production-Ready Microservices

How do we know we can trust a service?

Consider having a checklist.

A checklist? Seriously?



http://atulgawande.com/book/the-checklist-manifesto/

You know who uses checklists?

Pilots. Surgeons.

Should be quantifiable and measurable.

“Fast” won’t cut it.

Stability.

Reliability.

Scalability.

Fault tolerance.

Performance.

Monitoring.

Documentation.

I know what some of you are thinking…

I don’t have time for all this.

We need to MOVE FAST.
And break things…

We're Agile. With a capital A.

How is your velocity when an outage brings your business to a halt?

Requires buy in from the grass roots level as well as management.

Stable and Reliable.

Our services are evolving. Constantly.

Feature not a bug.

Development moves quickly.

Always Be Releasing.

Bit like spinning plates…

Just takes one wobble to take down the entire system.

How do we make sure our services are stable and reliable?

Consistent development process.

Code passes new and existing tests before committing code.

Pair programming. Code reviews.

External, automated build system.

Build pipeline.

Feature toggles.

Thorough testing.

Linting, unit tests, regression tests, integration tests, etc.

Deployment pipeline.

Must have a standardized deployment process.

Staging, canary, blue/green deploys.

Find issues *before* our service hits production servers.

Staging - exact copy of production.

Real world without real world traffic.

Capacity is often a percentage of production.

Though some organizations have identical prod/staging hardware.

Staging is the real deal.

Code should have already passed all our internal checks.

Unit tests, linting, QA,
customer acceptance etc.

If it checks out in staging, it is going to canary - with real traffic.

Canary - aka the canary
in the coal mine.

Find out if we have issues before we do a full production push.

Some percentage of production - 5% or 10%.

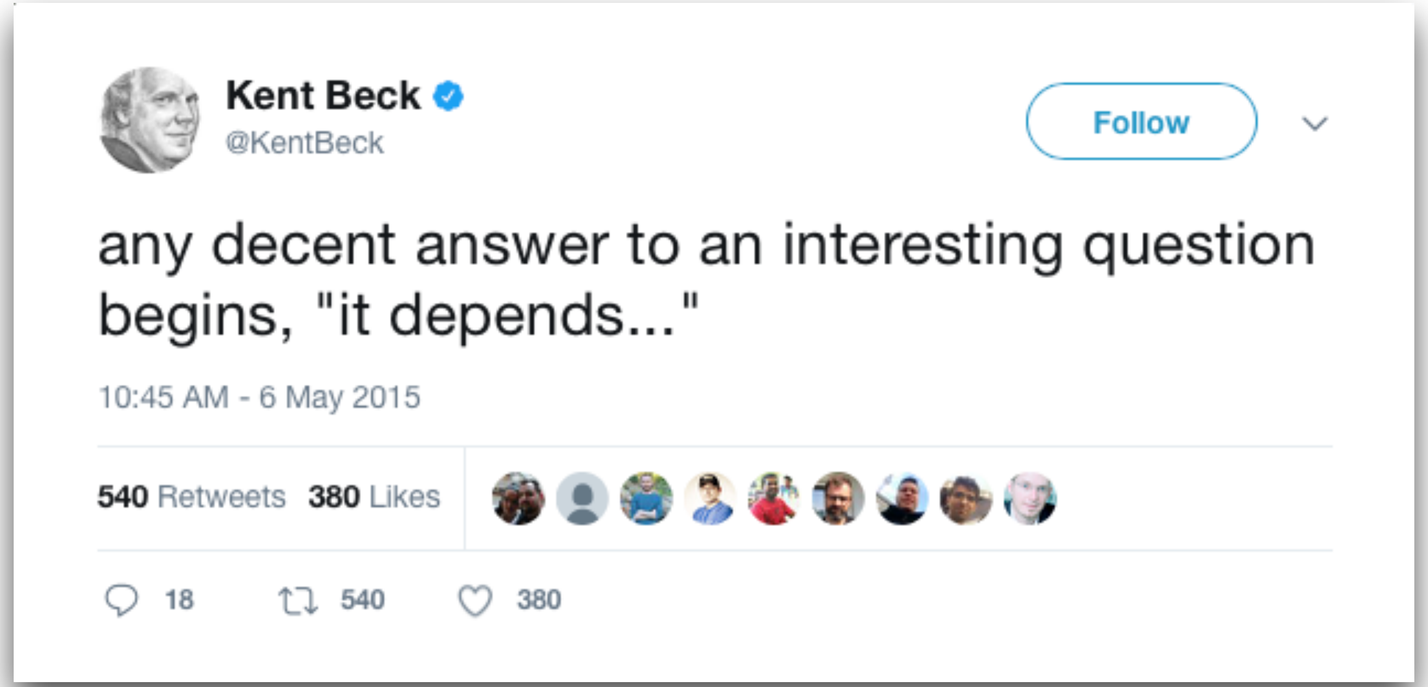
Can be a sliding scale too - start with 5%, move up to 20% etc.

Canaries are serving real production traffic.

Find errors? Automated rollbacks.

How long should our canary stage last? As long as it takes. Hours. Days.

¯_(ツ)_/¯



https://twitter.com/KentBeck/status/596007846887628801

How about replacing merge on GitHub.

On the fly.

Move Fast and Fix Things

vmg December 15, 2015

Anyone who has worked on a large enough codebase knows that technical debt is an inescapable reality: The more rapidly an application grows in size and complexity, the more technical debt is accrued. With GitHub's growth over the last 7 years, we have found plenty of nooks and crannies in our codebase that are inevitably below our very best engineering standards. But we've also found effective and efficient ways of paying down that technical debt, even in the most active parts of our systems.

At GitHub we try not to brag about the "shortcuts" we've taken over the years to scale our web application to more than 12 million users. In fact, we do quite the opposite: we make a conscious effort to study our codebase looking for systems that can be rewritten to be cleaner, simpler and more efficient, and we develop tools and workflows that allow us to perform these rewrites efficiently and reliably.

As an example, two weeks ago we replaced one of the most critical code paths in our infrastructure: the code that performs merges when you press the Merge Button in a Pull Request. Although we routinely perform these kind of refactorings throughout our web app, the importance of the merge code makes it an interesting story to demonstrate our workflow.

Merges in Git

We've talked at length in the past about the storage model that GitHub uses for repositories in our platform and our Enterprise offerings. There are many implementation details that make this model efficient in both performance and disk usage, but the most relevant one here is the fact that repositories are always stored *"bare"*.

This means that the actual files in the repository (the ones that you would see on your working directory when you clone the repository) are not actually available on disk in our infrastructure: they are compressed and delta'ed inside packfiles.

Because of this, performing a merge in a production environment is a nontrivial endeavour. Git knows several merge strategies, but the recursive merge strategy that you'd get by default when using `git merge` to merge two branches in a local repository assumes the existence of a working tree for the repository, with all the files checked out on it

How much do you agonize over a CSS change?

At this point, we should have very few production issues.

Most likely, we've found the problems in staging, canary, etc.

Your deployment pipeline should be *the* path to production. Period.

Avoid the temptation to perform hotfixes.

Bypassing the initial phases of the deployment pipeline often introduces new bugs into production, as emergency code fixes run the risk of not being properly tested.

— Susan J. Fowler
Production-Ready Microservices

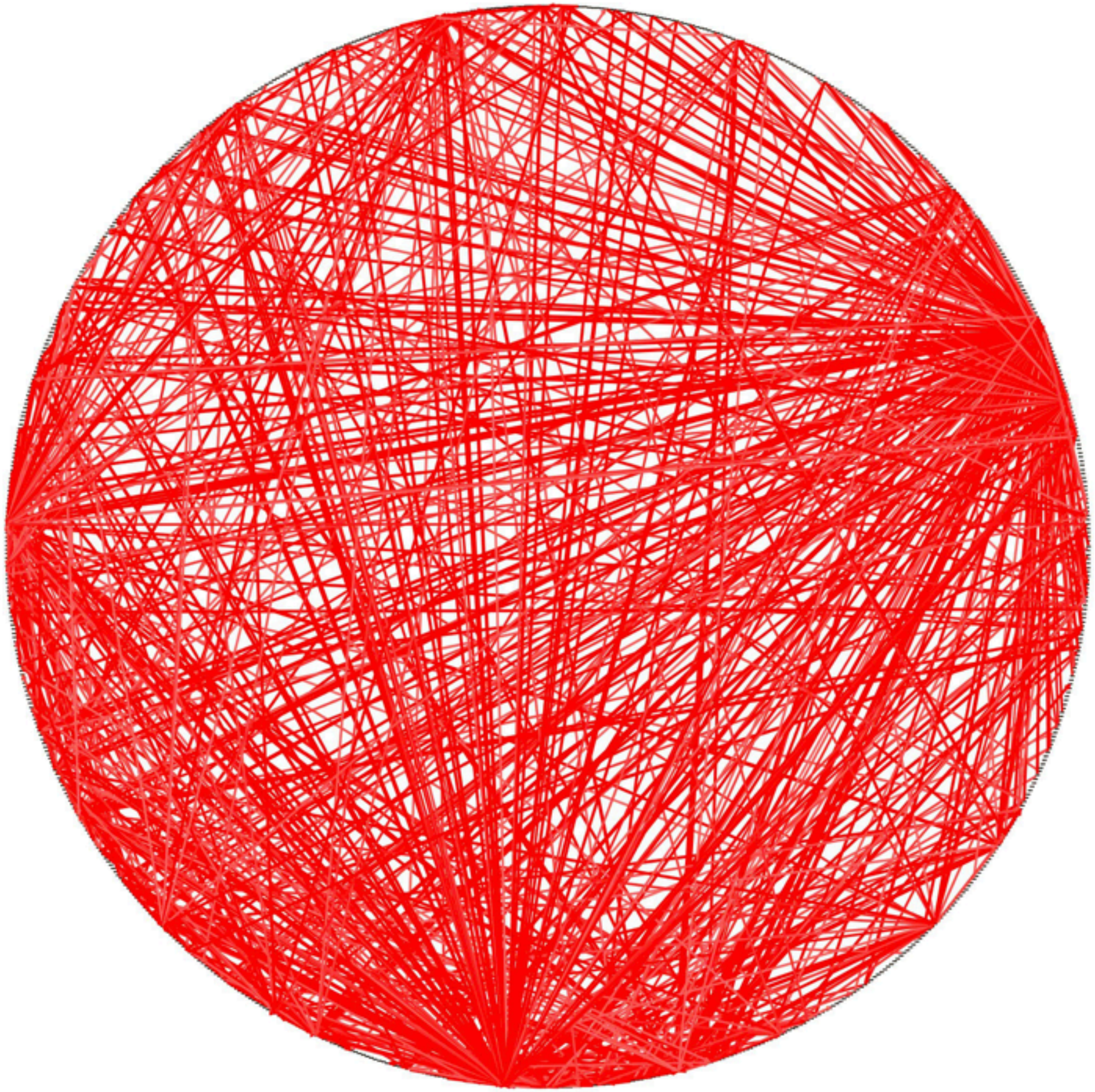
I’ve seen that movie…

Rollback to the last stable build.

Must also understand our dependencies.

And plan for their inevitable failure.

Dependency graphs get kind of crazy.



http://evolutionaryarchitecture.com

“Death Star Architecture.”

https://www.slideshare.net/adriancockcroft/goto-berlin

How do we mitigate a dependency failure?

Caching. Alternative services. Backups.

Depends on the criticality.

Submit a claim during a natural disaster.

Better be able to accept the claim.

But we might not work
on it immediately.

Maybe we need to post messages to a queue, process when we're up.

Our services should
have a health check.

Part V. Spring Boot Actuator: Production-ready features

Spring Boot includes a number of additional features to help you monitor and manage your application when it's pushed to production. You can choose to manage and monitor your application using HTTP endpoints, with JMX or even by remote shell (SSH or Telnet). Auditing, health and metrics gathering can be automatically applied to your application.

Actuator HTTP endpoints are only available with a Spring MVC-based application. In particular, it will not work with Jersey unless you enable Spring MVC as well.

46. Enabling production-ready features

The `spring-boot-actuator` module provides all of Spring Boot's production-ready features. The simplest way to enable the features is to add a dependency to the `spring-boot-starter-actuator` 'Starter'.

> **Definition of Actuator**
>
> An actuator is a manufacturing term, referring to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

To add the actuator to a Maven based project, add the following 'Starter' dependency:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

For Gradle, use the declaration:

```
dependencies {
    compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

47. Endpoints

Actuator endpoints allow you to monitor and interact with your application. Spring Boot includes a number of built-in endpoints and you can also add your own. For example the `health` endpoint provides basic application health information.

The way that endpoints are exposed will depend on the type of technology that you choose. Most applications choose HTTP monitoring, where the ID of the endpoint is mapped to a URL. For example, by default, the `health` endpoint will be mapped to `/health`.

The following technology agnostic endpoints are available:

Always Be (health) Checking.

Not healthy? Don’t route traffic to it.

Circuit breakers for fun and profit.

GitHub, Inc.

HYSTRIX

DEFEND YOUR APP

Hystrix: Latency and Fault Tolerance for Distributed Systems

OSS Lifecycle inaccessible | build failing | License Apache 2

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

Full Documentation

See the Wiki for full documentation, examples, operational details and other information.

See the Javadoc for the API.

Communication

- Google Group: HystrixOSS
- Twitter: @HystrixOSS
- GitHub Issues

What does it do?

1) Latency and Fault Tolerance

Stop cascading failures. Fallbacks and graceful degradation. Fail fast and rapid recovery.

Thread and semaphore isolation with circuit breakers.

2) Realtime Operations

Defend the neighborhood as it were.

Services won’t live forever.
Plan for decommissioning.

Need a deprecation period.

Typically more a cultural issue than a technical one.

Monitoring is key - are we still seeing calls to the old endpoint?

Don’t just abandon a service!

Scalable.

We need to know the growth scale of our service.

How many requests can we handle per second? RPS.

We can look at current load levels.

We can (and should!)
perform load tests.

Potential fitness function…

Very important but we need to look beyond just our service.

A given service is just one part of a larger world.

What is the qualitative growth scale?

Linked to our business.

Do we need to scale by the number of users? Number of orders?

Isn’t tied to a particular service!

We will need to talk to our business partners to understand the drivers.

We will have to translate
those to our specific services.

Accurate growth scales are vital.

All services are equal. Some services are more equal than others.

Business critical services should have priority.

Resource isolation is your friend…
shared hardware hurts.

May want to engage in some capacity planning.

Ensure we have sufficient budget.

Or change your process…

Don’t neglect your dependencies.

Can they scale with you?

Better make sure they can!

Collaborate. Communicate.

Have a scalability review with the services you rely on.

What does your data growth look like?

What type of database makes the most sense? Relational? NoSQL?

Is eventual consistency ok?

Are you read heavy? Write heavy?

How do we scale our database?

`cf scale FOO` doesn’t exempt us from thinking through scalability.

Fault Tolerant.

We can’t prevent catastrophes. But we can build failure resistant services.

Hope for the best…
prepare for the worst.

ROBERT MCMILLAN BUSINESS 10.13.15 7:00 AM

HER CODE GOT HUMANS ON THE MOON—AND INVENTED SOFTWARE ITSELF

Hamilton wanted to add error-checking code to the Apollo system that would prevent this from messing up the systems. But that seemed excessive to her higher-ups. “Everyone said, ‘That would never happen,’” Hamilton remembers.
But it did. Right around Christmas 1968.

— ROBERT MCMILLAN

https://www.wired.com/2015/10/margaret-hamilton-nasa-apollo/

Failures, uh find a way.

Eliminate single points of failure!

Not sure what they might be?

Draw up the architecture.

What happens if *this* fails?

It can’t fail? Yeah it can - what happens if it does?

Think through how
our service could fail.

It is hard. We are really good at thinking through the happy path.

But we need to think about the road less traveled.

Test for these failure scenarios. Does our service respond appropriately?

Only one way to really know…

NETFLIX

SIMIAN ARMY

https://github.com/Netflix/SimianArmy

Chaos engineering.

http://principlesofchaos.org

Intentionally break things.

Add latency.

Shut down an availability zone.

Kill a machine.

Shouldn’t be ad hoc! Monitor and log.

Chaos can go rogue…

Some things will fall through the proverbial cracks.

Need to detect failures.

And add them to our testing suite.

Failures could be internal or external.

How is your exception handling?

Code will have bugs. Test thoroughly.

Extremely common for a dependency to cause our service to fail!

Downstream service…or a 3rd party library.

How does your service recover from a failure?

Resiliency testing.

Make your service fail. Repeatedly.

Load testing. But we covered that!

Where do you run your load tests?

Staging?

Good place to start…however it isn't the same as testing in production.

QA in Production

04 April 2017

Rouan Wilsenach

Rouan is a software developer. He spent a number of years working as a consultant, including at ThoughtWorks, developing applications for clients in the financial services, health, media and education sectors. He now works remotely for Tes, where he builds software to help teachers and schools. He likes writing and speaking at conferences.

Find **similar articles** to this by looking at these tags: continuous delivery · testing

Contents

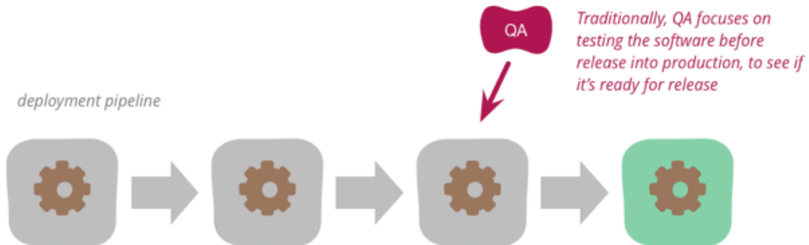
Gathering production data
- Critical success indicators
- Logging
- Metrics
- APIs

Learning from production data
- Alerting
- Dashboards

A QA approach rooted in reality
- Tests need to earn their keep
- Are you ready to adopt production QA practices?
- Finding the right balance

Gathering operational data about a system is common practice, particularly metrics that indicate system load and performance such as CPU and memory usage. This data has been used for years to help teams who support a system learn when an outage is happening or imminent. When things become slow, a code profiler might be enabled in order to determine which part of the system is causing a bottleneck, for example a slow-running database query.

I've observed a recent trend that combines the meticulousness of this traditional operational monitoring with a much broader view of the quality of a system. While operational data is an essential part of supporting a system, it is also valuable to gather data that helps provide a picture of whether the system as a whole is behaving as expected. I define "QA in production" as an approach where teams pay closer attention to the behaviour of their production systems in order to improve the overall quality of the function these systems serve.



…an ecommerce company that accidentally ran its tests against its production ordering systems. It didn't realize its mistake until a large number of washing machines arrived at the head office.

— Sam Newman
Building Microservices

Synthetic transactions.

And be sure everyone knows when you're doing it.

Especially downstream dependencies!

Load testing shouldn’t be a one off.

Can be automated and run on a cadence.

Probably not during your peek traffic hours mind you…

Ultimate goal - reduce the impact on our users.

Roll back.

Fail over.

Automate, automate, automate.

All services are equal. Some services are more equal than others.

What is the impact of
this service failing?

Categorize incidents and outages by severity and scope.

Severity - impact to the business.

Scope - how much of the business is impacted?

Monitored.

Four components to monitoring.

Logging.

What would you say
my service is doing?

Log anything that is useful.

Just don’t put in any personally identifying information (PII).

Ever.

Some things alone aren't PII but when combined with other items…

Tracing can be difficult.
Correlation IDs help.

Dashboards.

View the health of a service.

More on this in a minute!

Alerting.

A key metric is out of band.

Allows us to detect an issue and fix it before our customers even notice.

Pager duty.

Must be sustainable.

Provide clear, concise on call documentation.

Vital that we think about just what we should be monitoring.

What *is* a key metric?

Some pertain solely to the infrastructure our service runs on.

CPU utilization, RAM utilization, threads, database connections…

These often impact more than just our service.

Others key metrics are specific to our service.

Additionally we need to know the availability, latency, response time…

Basically anything that we identified earlier as part of our SLO.

Monitor errors and exceptions as well.

Identify normal, warning and critical thresholds for your metrics.

Can be hard to figure out early on.
Need a certain amount of history.

Not just a prod thing. We need to monitor staging. Validates the monitors.

Metrics should be displayed on a dashboard of some sort.

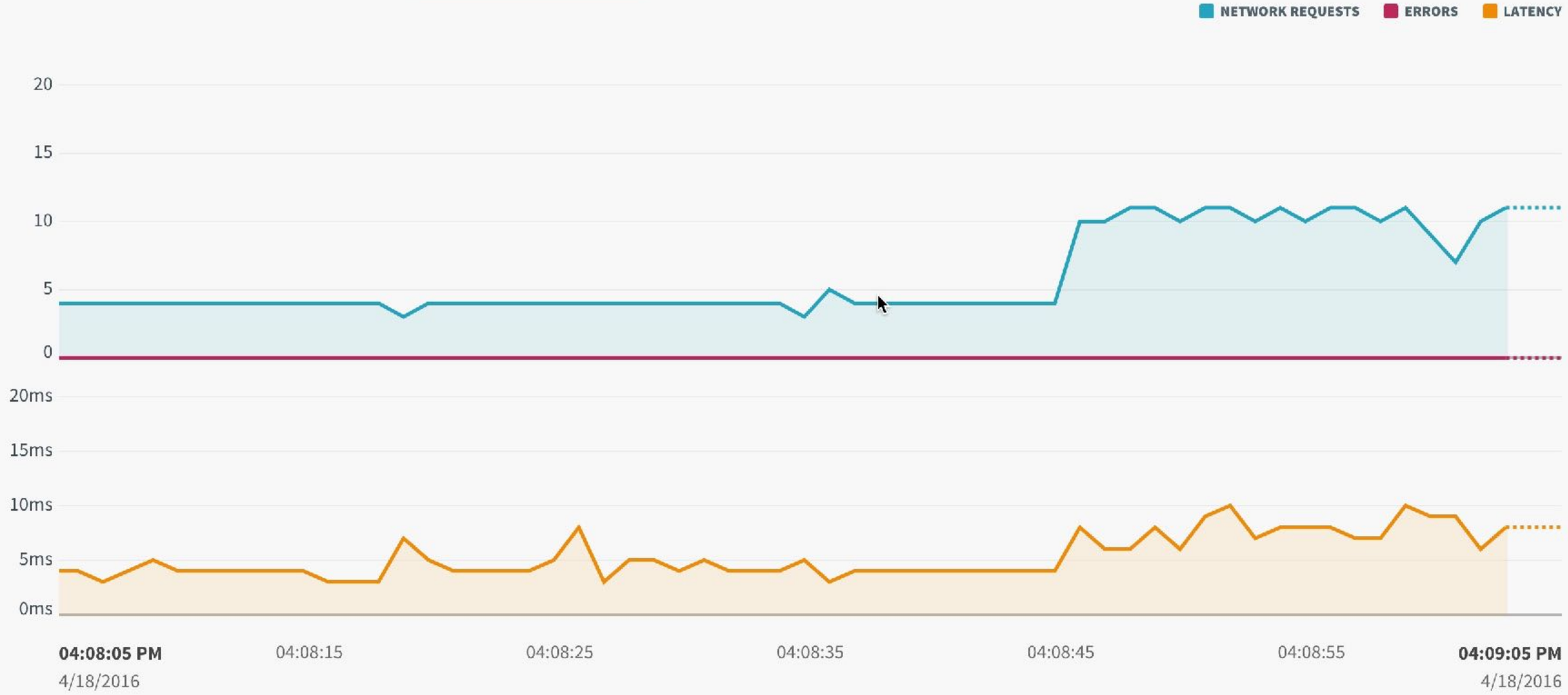
PCF Metrics | fortune-ui

https://metrics.run.pivotal.io/apps/ff713209-ad5d-4b6e-bfe1-0829d7f3ac68/network

fortune-ui

ORG: **platform-eng** SPACE: **nfjs-workshop** STATUS: **Running**

DASHBOARD | CONTAINER METRICS | NETWORK METRICS



metrics.run.pivotal.io

PCF Metrics

Search for Application...

Feedback NSCHUTTA@PIVOTAL.IO

spo-ring-todo ORG: app[0] SPACE: spo-ring-homework STATUS: Running

Oct 4, 9:16 am — Oct 5, 9:16 am (local)

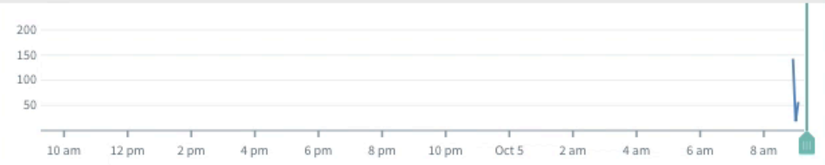
Sep 22 Sep 23 Sep 24 Sep 25 Sep 26 Sep 27 Sep 28 Sep 29 Sep 30 Oct 1 Oct 2 Oct 3 Oct 4 Oct 5

DASHBOARD

view instances

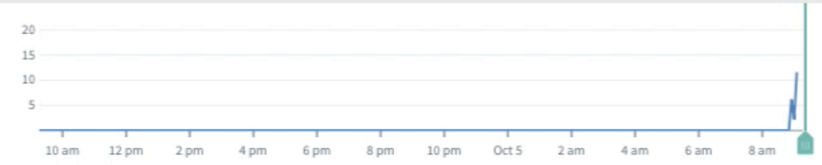
+ ADD CHART

cf.system.latency



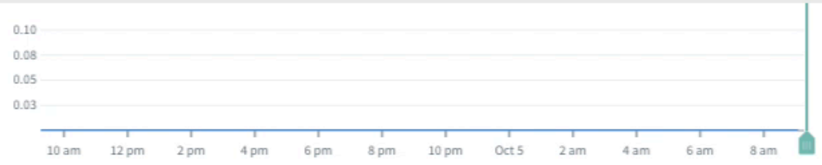
gauge | average across instances | milliseconds

cf.system.request-count



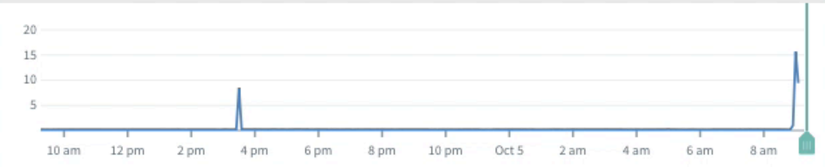
gauge | total across instances | per minute

cf.system.request-error-count



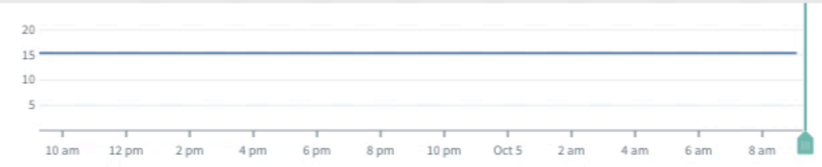
gauge | total across instances | per minute

cf.system.cpu



gauge | average across instances | percentage

cf.system.disk

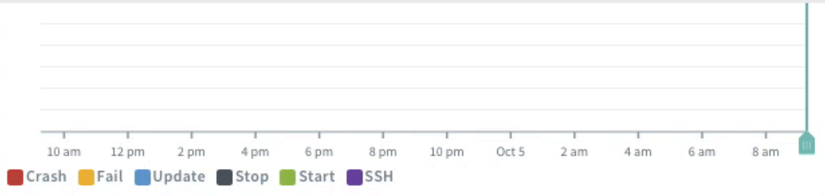


gauge | average across instances | percentage

cf.system.memory

gauge | average across instances | percentage

cf.system.events



LOGS

A Medium Corporation

mattklein123 Follow
Engineer @lyft
Oct 25 · 4 min read

Lyft's Envoy dashboards

Front/edge Envoy per-host row

I've given quite a few talks about observability in the age of the service mesh (most recent slides, unfortunately this talk series has not been recorded yet). Visibility into the inherently unstable network is one of the most important thing that Envoy provides and I'm asked repeatedly for the source of the dashboards that we use at Lyft. In the interest of "shipping" and getting something out there that can help folks, we are releasing a snapshot of our internal Envoy dashboards.

What we are releasing is unfortunately *not* going to be readily consumable. It is also not an OSS project that will be maintained in any way. The goal is to provide a snapshot of what Lyft does internally (what is on each dashboard, what stats do we look at, etc.). Our hope is having that as a reference will be useful in developing new dashboards for your organization.

107 1

Next story
Coworkers Will Become Custo...

But we should be alerted when things start to go wonky.

We shouldn’t be staring at our dashboards all day!

Alert on all of our key metrics, SLOs etc.

Absence of a key metric is also an avertable offense!

Alerts should be actionable.

Create an on call book to assist team monitoring the service.

How to mitigate, resolve, etc.

Step by step instructions.
Do not make asumptions.

Follow the 3 AM Rule.

Write it assuming the person reading will be half asleep.

Do not be clever. Think simple.

We don't rise to the level of our expectations, we fall to the level of our training.

— Archilochus

This all implies an on call rotation.

Sorry.

Everyone should do it. Rotations need to be sustainable.

Pair up.

One week or less with at least a month off in between.

One developer on call for 3 years?

Not so much.

Documented.

What does your service do?

How does it work?

What does it depend on?

Ever say something like “the documentation is useless”?

It doesn’t have to be.

Golden rule!

Do it for those that come after you.

Don’t forget, sometimes *you* are the person that comes after you!

How long does it take for a new team member to be productive? Weeks?

Months?

Solid onboarding guide.

Make sure it is updated.

Documentation should be easy to find.

Probably a website/wiki.

Updating the wiki should be a normal part of the developer workflow.

Consider a simple (low ceremony) template.

Description - what does your service do? Don’t skimp here.

An architectural diagram or three.

Contact information as
well as the on call rotation.

Links to helpful things like the repo, dashboard link, on call book.

FAQ.

Onboarding/development guide.

Coding standards.

Development pipeline.

Whatever helps the team understand.

Everyone should “get it” and be able to describe it. So have them do it.

Shouldn’t be a static thing!

Documentation should be reviewed along with the architecture.

Production Readiness Reviews.

Not a one time, up front thing.

Services should be reviewed and audited regularly.

Does not have to be high ceremony!

Get the team together - SREs, Devs, etc. Draw up the architecture.

Do we have a shared understanding of what the system does?

Do we have a shared understanding of our requirements?

As we talk through it, we will discover bottlenecks.

The Wombat service has a lower availability level than we need.

We will find interesting failure cases.

“When month end falls on the Harvest Moon.”

Review should result in a new architecture diagram or two.

And probably some new items on the backlog.

Perform an audit.

Go back to your checklist. Does the service meet our requirements?

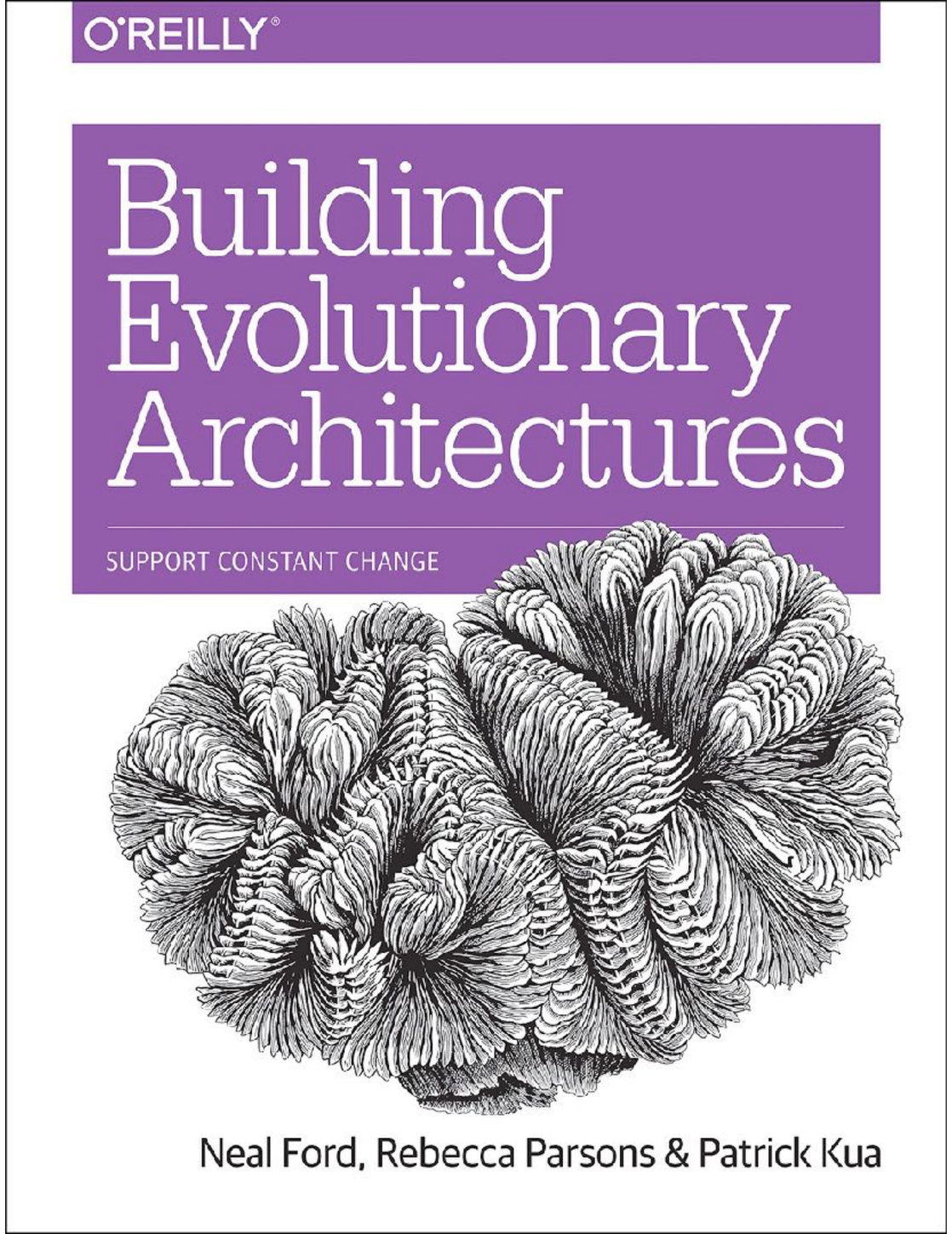
Probably results in some new things in our backlog!

Now we can create a production readiness roadmap.

What do we need to fix and when can/should we fix it.

Drive prioritization of the work.

A lot of this is manual. But some of it can be automated!



http://evolutionaryarchitecture.com

Fitness functions!

Basically, a set of tests we execute to validate our architecture.

How close does this particular design get us to our objectives?

Ideally, all automated. But we may need some manual verifications.

For example…

All service calls must respond within 100 ms.

Cyclomatic complexity shall not exceed X.

Hard failure of an application will spin up a new instance.

Reviews and audits should not be additional red tape.

Should not be overly bureaucratic.

Couple of hours…depending on the complexity of the system.

Next steps.

Check your culture.

Work on that checklist.

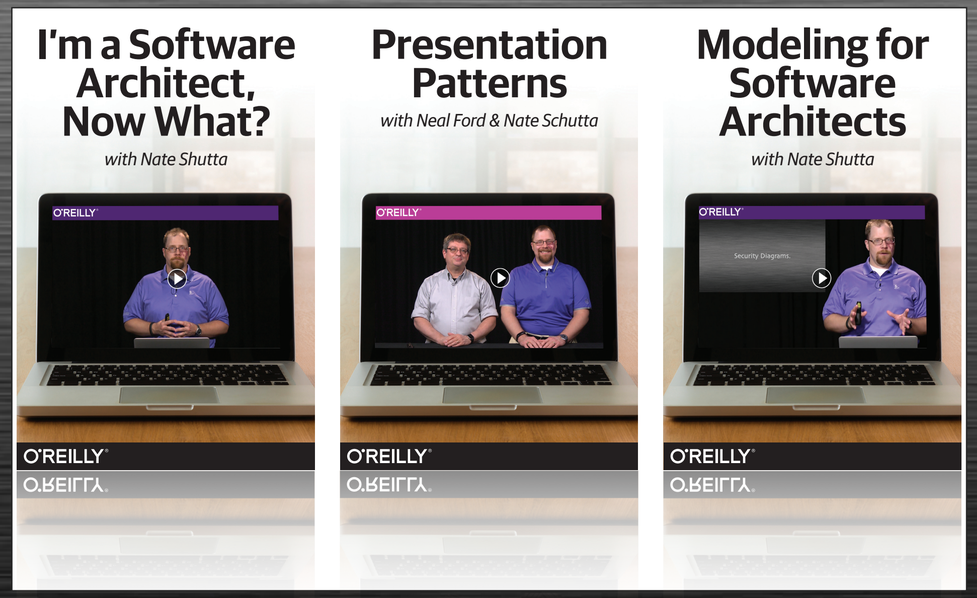
Review some services!

Adapt and change.

Good luck!

Questions?

Thanks!



Nathaniel T. Schutta
@ntschutta