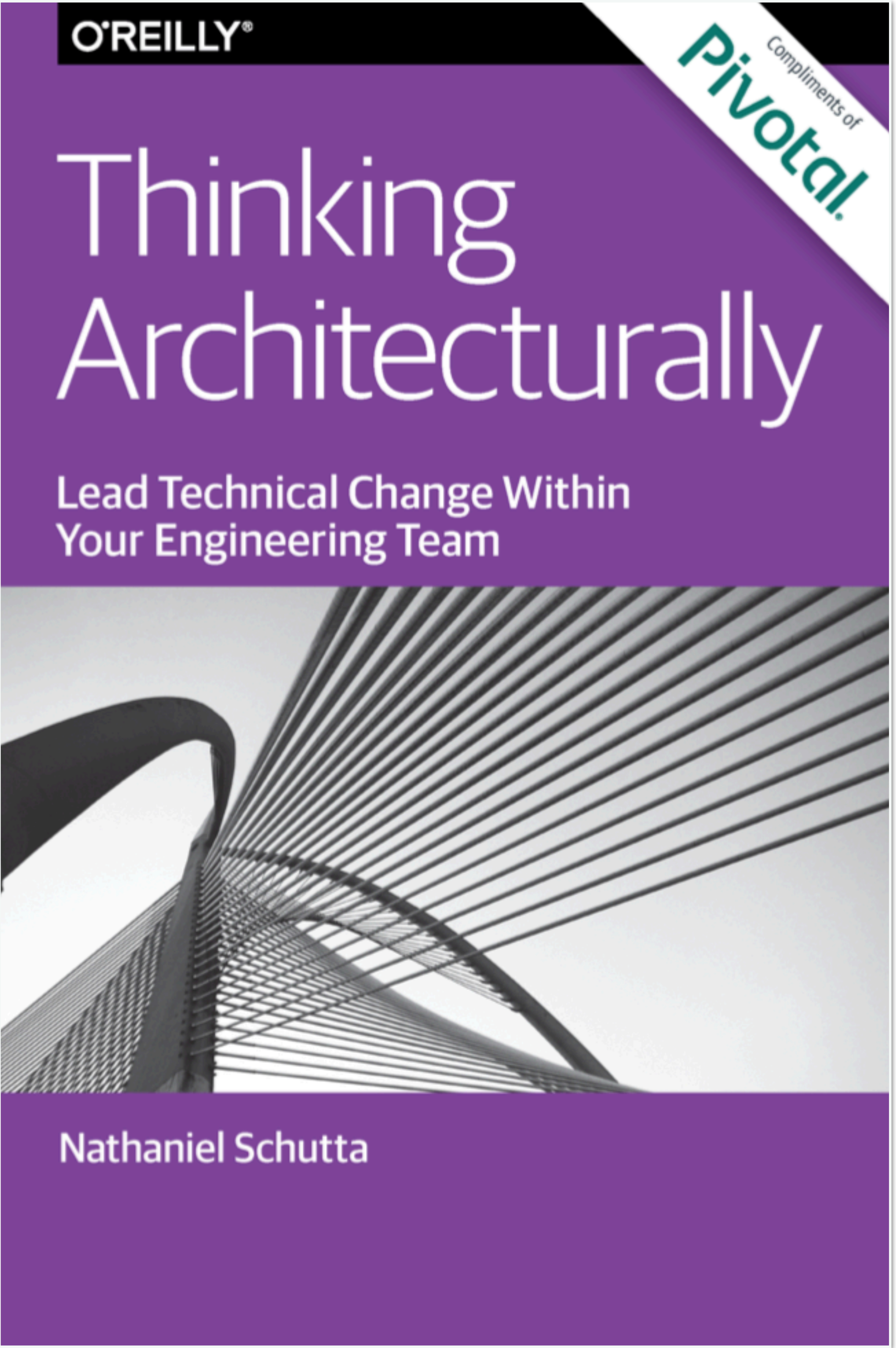
# Responsible Microservices

Nathaniel Schutta
@ntschutta
ntschutta.io


O'REILLY
Compliments of Pivotal
# Thinking Architecturally
Lead Technical Change Within Your Engineering Team
Nathaniel Schutta

https://content.pivotal.io/ebooks/thinking-architecturally

Ah “the cloud!”

So. Many. Options.

For better or worse…

Many developers think:
cloud === microservices.

Lot of time spent on domain driven design.

## Looking for bounded contexts…

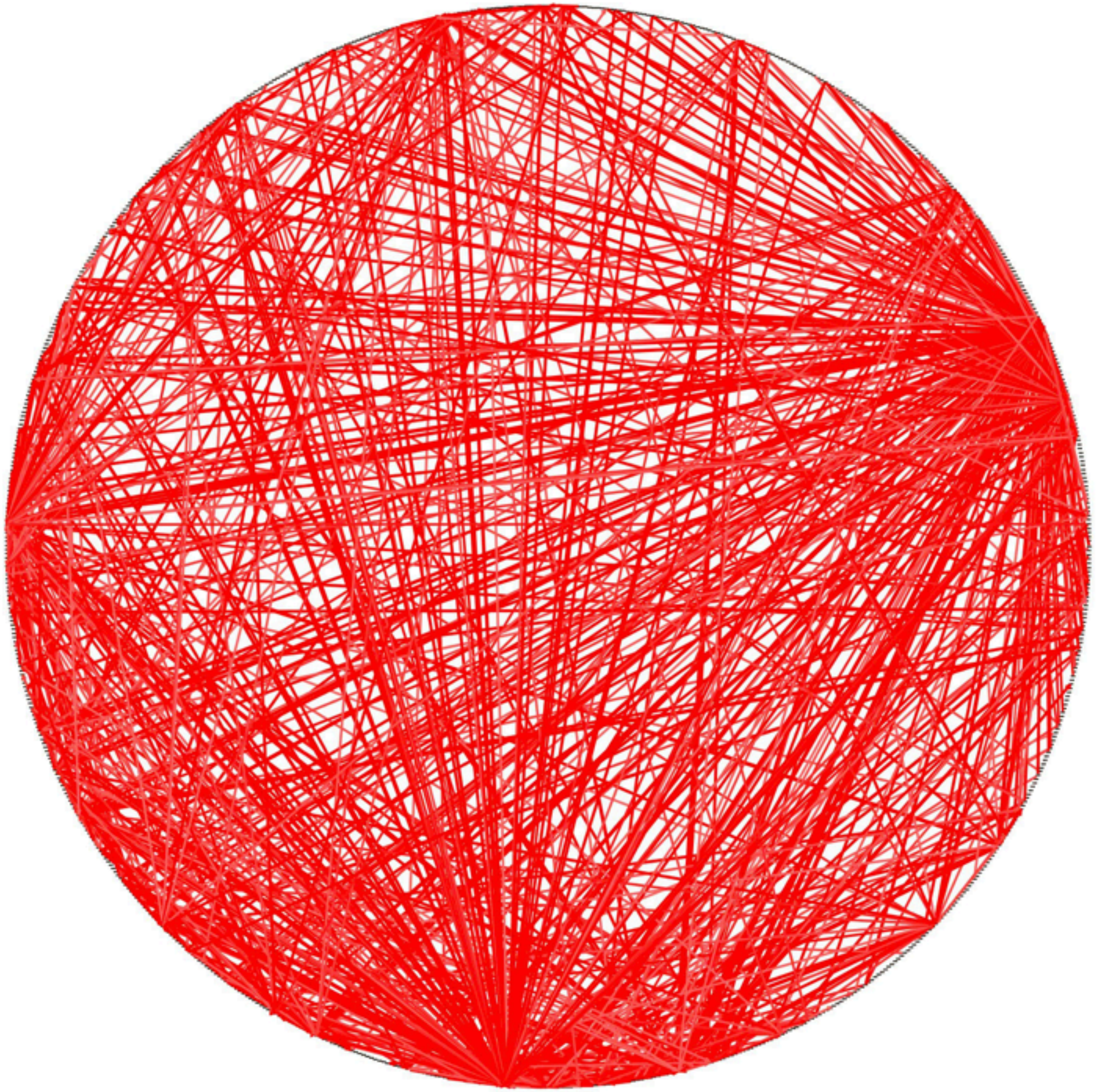
## Defining a ubiquitous language.

## Forming two pizza teams.

“Your [developers] were so preoccupied with whether they could, they didn't stop to think if they should.”

–Ian Malcolm

And that has led to some serious pain.



http://evolutionaryarchitecture.com

Oops.

There are a number of good reasons to adopt microservices.

But there are no free lunches.

We have to consider the cost of the added complexity.

Does your application actually benefit?

Or are you just adding accidental complexity?

## When should we consider microservices?

## Please Microservice Responsibly.

https://content.pivotal.io/blog/should-that-be-a-microservice-keep-these-six-factors-in-mind

# MICROSERVICES

Reaction to monoliths and heavy weight services.

As well as cloud environments.

Monoliths hurt.

Developer productivity takes a hit.

Hard to get your head wrapped around a huge code base.

Long ramp up times
for new developers.

Small change results in building and deploying everything.

Scaling means scaling the entire application!

Not just the part that needs more capacity.

Hard to evolve.

We’re all familiar with the second law of thermodynamics…

Otherwise known as a teenagers bedroom.

The universe really
wants to be disordered.

Software is not immune
from these forces!

Modularity tends to
break down over time.

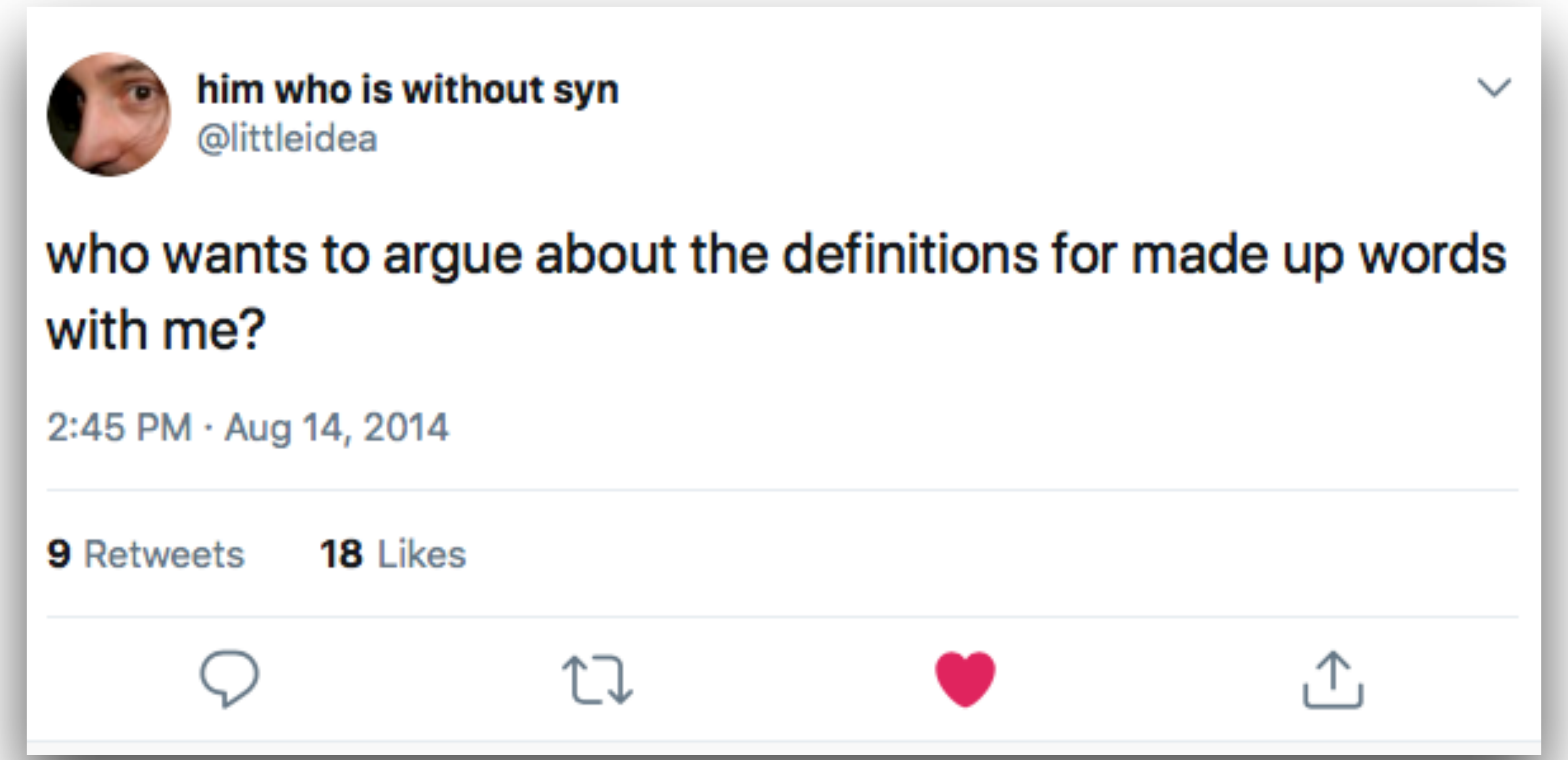
Over time, takes longer to add new functionality.

Frustration has given birth to a “new” architectural style.

## Enter the microservice.

No “one” definition.

In the eye of the beholder…


him who is without syn
@littleidea

who wants to argue about the definitions for made up words with me?

2:45 PM · Aug 14, 2014

9 Retweets 18 Likes

https://mobile.twitter.com/littleidea/status/500005289241108480

Anything that can be rewritten two weeks or less.

Think in terms of characteristics.

Suite of small, focussed services.

Do one thing, do it well.

Linux like - pipe simple things together to get complex results.

Independently deployable.

Independently scalable.

Evolve at different rates.

Freedom to choose the right tech for the job.

Built around business capabilities.

High cohesion, low coupling…

Applied to services.

It is just another approach. An architectural style. A pattern.

Despite what some developers may have said.

Use them wisely.

“If you can't build a monolith, what makes you think microservices are the answer?”

–Simon Brown

http://www.codingthearchitecture.com/2014/07/06/distributed_big_balls_of_mud.html

Sometimes the right answer is a modular monolith…

https://www.youtube.com/watch?v=kbKxmEeuvc4

# Multiple Rates of Change

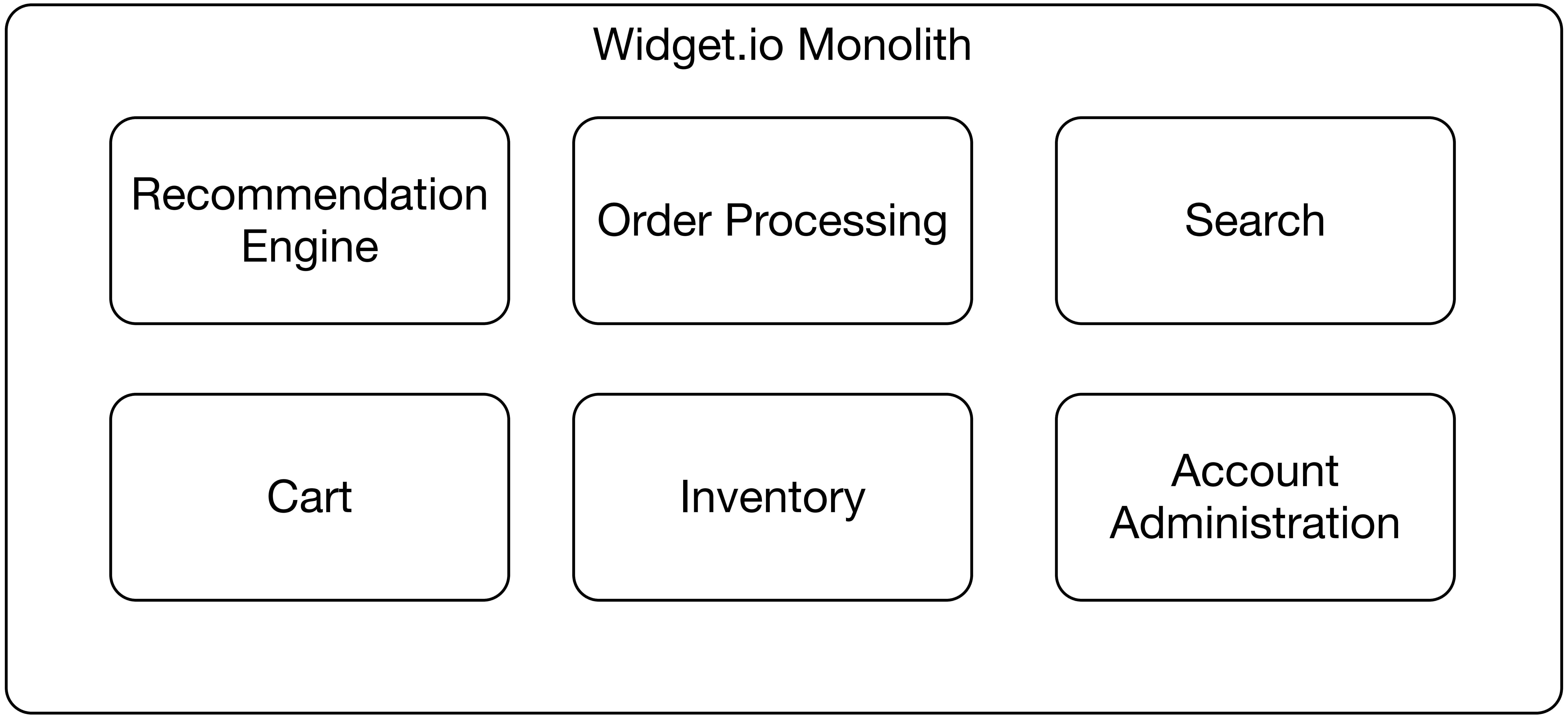
Some parts of your system change all the time.

Others haven’t changed in months. Or years.

If parts of your system evolve at different speeds…

## You might need microservices!

For your consideration…the Widget.io Monolith!


Widget.io Monolith
Recommendation Engine
Order Processing
Search
Cart
Inventory
Account Administration

The Cart module probably doesn’t change much.

Maybe the Inventory system is really stable.

But our product owners constantly tweak the Recommendation Engine.

And we are always improving Search.

In a monolith, everything has to move at the same rate.

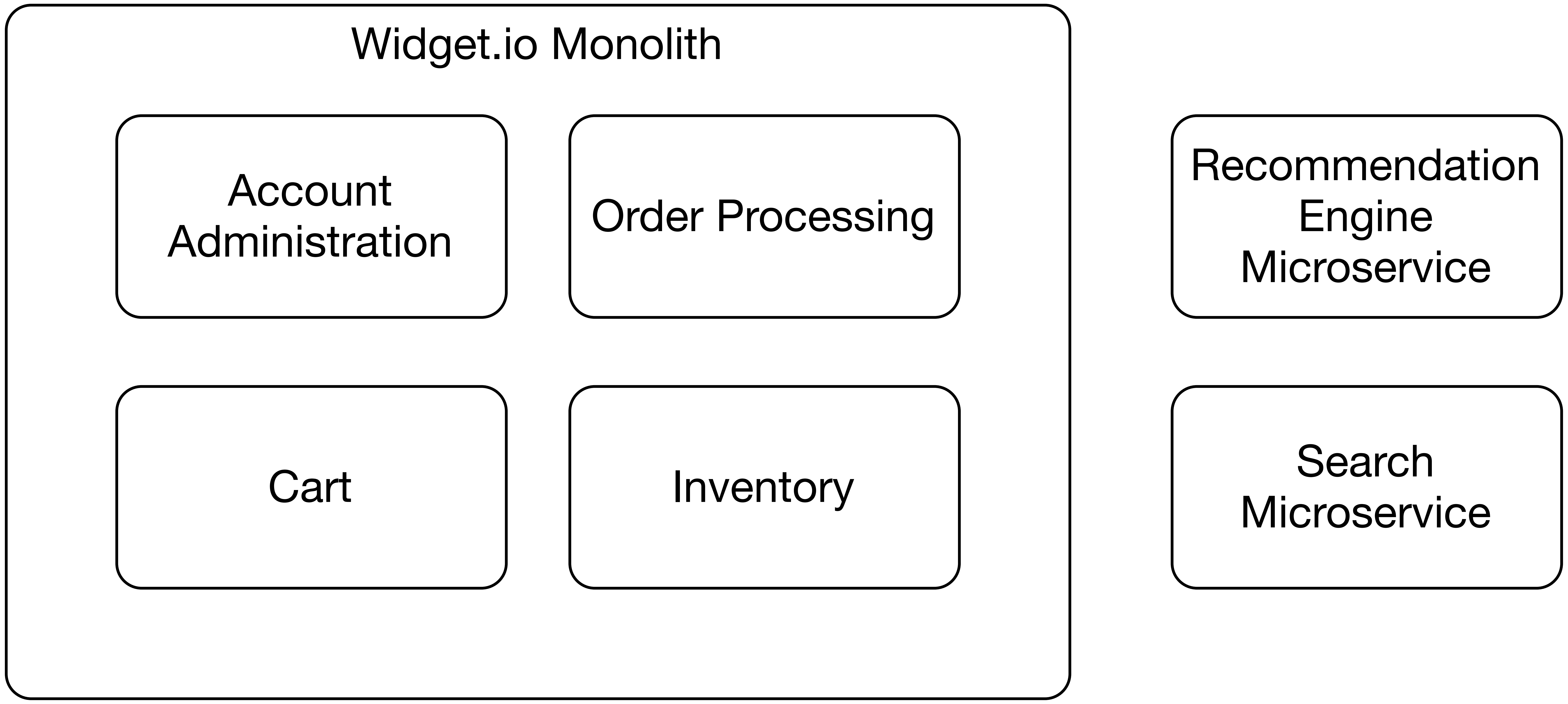
## Why a quarterly release?

Because that is when **all** the changes were ready.

And since we had to push the entire monolith anyway…

Today we have options.

Splitting them out allows us to iterate those features faster.


Widget.io Monolith
Account Administration
Order Processing
Cart
Inventory
Recommendation Engine Microservice
Search Microservice

Enables us to deliver business value quickly.

How do we find the components that change far faster than the rest?

You probably have an inkling already in your systems.

Trust your gut instincts!

But it can be very helpful to have, well, some data.

Start with your source code management tool…

You can get a “heat map” of sorts just by looking at history.

```
git log --pretty=format: --name-only |
 sort | uniq -c | sort -rg | head -10
```

Running that against Spring…

(Not a monolith but for pedagogical reasons…)

```
12. nschutta@Firethorn: ~/work/spring/spring-framework (zsh)
spring-framework (master) » git log --pretty=format: --name-only | sort | uniq -c
| sort -rg | head -10
15983
 991 build.gradle
 239 src/asciidoc/index.adoc
 187 build-spring-framework/resources/changelog.txt
 129 spring-core/src/main/java/org/springframework/core/annotation/AnnotationUtils
.java
 119 src/dist/changelog.txt
 106 spring-beans/src/main/java/org/springframework/beans/factory/support/DefaultL
istableBeanFactory.java
  96 spring-webmvc/src/main/java/org/springframework/web/servlet/config/annotation
/WebMvcConfigurationSupport.java
  94 spring-context/src/main/java/org/springframework/context/annotation/Configura
tionClassParser.java
  94 org.springframework.core/src/main/java/org/springframework/core/convert/TypeD
escriptor.java
spring-framework (master) »                    ~/work/spring/spring-framework
```

Gives you a place to start.

## Software archeology time!

Roll up your sleeves and root around your codebase.

Look for (apologies Isaac Newton) smoother pebbles and prettier shells.

Look for what Michael Feathers coined "churn".

https://www.stickyminds.com/article/getting-empirical-about-refactoring

## Where should we refactor?

When you look at your project,
there will be a “long tail.”

Some files are updated constantly, others just initial commit.

Chad Fowler created Turbulence based on churn vs. complexity.

https://github.com/chad/turbulence

There are “code forensic” tools we can leverage as well.

CodeScene.

https://codescene.io/about

codescene.io

CODESCENE CLOUD

Plans Documentation Log in

ANALYSIS RESULTS

DOCKER

Dashboard
Scope
Technical Debt
Architecture
Social Analyses
Project Management

SCOPE

224,787 Lines of Code

185,334 Lines of Go

1270 Authors

56 Active Authors

16289 Commits

HOTSPOTS

5.5%
Red Hotspots

14.3%
Development Effort in Red Hotspots

29%
of Estimated Bugfixes in the Hotspots

CODE BIOMARKERS

D
Current Indication

D
Last Month

D
Last Year

AUTHORS

0.0 Months
Median Contribution

57 Months
Longest Contribution

About FAQ Blog Showcase Pricing Support Copyright © 2015-2019 Empear

codescene.io
CODESCENE CLOUD
Plans
Documentation
Log in
ANALYSIS RESULTS
DOCKER
Dashboard
Scope
Technical Debt
Hotspots
Code Biomarkers
Refactoring Targets
Temporal Coupling
Architecture
Social Analyses
Project Management
Hotspots
Refactoring Targets
Code Age
Defects
Programming Language
System
docker
docker
About
FAQ
Blog
Showcase
Pricing
Support
Copyright © 2015-2019 Empear

Hotspots identify the modules with most development activity -- often technical debt.

codescene.io
CODESCENE CLOUD
Plans
Documentation
Log in
ANALYSIS RESULTS
DOCKER
Dashboard
Scope
Technical Debt
Hotspots
Code Biomarkers
Refactoring Targets
Temporal Coupling
Architecture
Social Analyses
Project Management

Hotspots identify the modules with most development activity -- often technical debt.

Hotspots
Refactoring Targets
Code Age
Defects
Programming Language
stream
state.go
container_windows.go
health.go
archive.go
monitor.go
store.go
container_unix.go
memory_store.go
view_test.go
env.go
env_test.go
memory_store_test.go
state_test.go
container_unit_test.go
view.go
container.go

System / docker / container

- stream
- archive.go
- container.go
- container_linux.go
- container_notlinux.go
- container_unit_test.go
- container_unix.go
- container_windows.go
- env.go
- env_test.go
- health.go
- history.go
- memory_store.go
- memory_store_test.go
- monitor.go
- mounts_unix.go
- mounts_windows.go
- state.go

codescene.io

CODESCENE CLOUD

Plans Documentation Log in

ANALYSIS RESULTS

DOCKER

Dashboard
Scope
Technical Debt
Hotspots
Code Biomarkers
Refactoring Targets
Temporal Coupling
Architecture
Social Analyses
Project Management

Hotspots identify the modules with most development activity -- often technical debt.

Hotspots | Refactoring Targets | Code Age | Defects | Programming Language

System / docker / container / container.go

| | |
|---|---|
| Size | 811 Lines of Code |
| Change Frequency | 804 Commits |
| Main Author | Michael Crosby (16 %) |
| Knowledge Loss | 0 % Abandoned Code |
| Defects | 184 (22 % Bug Fixes) |
| Last Modified | 0 months ago |

Code Biomarker D

Actions

View Code X-Ray Trends Authors

Complexity Trend

# X-Ray File Results

Projects / docker / Files / docker/container/container.go

Hotspots | Internal Temporal Coupling | Structural Recommendations

| Function | Change Frequency | Complexity/Size | Cyclomatic Complexity |
|---|---|---|---|
| BuildCreateEndpointOptions | 30 | 173 | 46 |
| SetupWorkingDirectory | 29 | 28 | 13 |
| StartLogger | 27 | 47 | 12 |
| GetResourcePath | 24 | 16 | 4 |
| ShouldRestart | 24 | 4 | 1 |
| AddMountPointWithVolume | 20 | 16 | 1 |
| NewBaseContainer | 20 | 11 | 1 |
| FromDisk | 19 | 28 | 8 |
| BuildJoinOptions | 17 | 17 | 6 |
| startLogging | 15 | 17 | 5 |
| StdoutPipe | 12 | 3 | 1 |
| StderrPipe | 12 | 3 | 1 |
| BuildEndpointInfo | 11 | 47 | 14 |

codescene.io
CodeScene
CodeScene
CODESCENE CLOUD
Plans
Documentation
Log in
ANALYSIS RESULTS
DOCKER
Dashboard
Scope
Technical Debt
Hotspots
Code Biomarkers
Refactoring Targets
Temporal Coupling
Architecture
Social Analyses
Project Management

# Complexity Trend

Hotspots / docker/container/container.go / Complexity Trend

Complexity (ws)
3,500
3,000
2,500
2,000
1,500
1,000
500
0
2014
2015
2016
2017
2018
Complexity
Lines of Code
Code Comments

## Descriptive Statistics

### Complexity vs. Lines of Code

About
FAQ
Blog
Showcase
Pricing
Support
Copyright © 2015-2019 Empear

codescene.io
CodeScene
CodeScene
CODESCENE CLOUD
Plans
Documentation
Log in
ANALYSIS RESULTS
DOCKER
Dashboard
Scope
Technical Debt
Architecture
Hotspots
Conway's Law
Temporal coupling
Social Analyses
Project Management
Authors
Teams
Knowledge Loss
Coordination Needs
Technical Sprawl
Package
Client
Integration CLI
Plugin
Layer
API
Container
Profiles
Lib Containerd
Hack
Run Config
Registry
CMD
Image
Builder
Volume
Distribution
Daemon
● Unassigned
● Docker
● Microsoft
Authors
Teams
About
FAQ
Blog
Showcase
Pricing
Support
Copyright © 2015-2019 Empear

Measure how well your organization aligns with your architecture.

System / Integration CLI

| | |
|---|---|
| **Size** | 34475 Lines of Code |
| **Primary Author** | Vincent Demeester (10 %) ● |
| **Primary Team** | Unassigned (66 %) ● |
| **Knowledge Loss** | 0 % Abandoned Code |
| **Team Fragmentation** | 0.48 (0.0 -> 1.0) |
| **Number of Teams** | 5 |

**Actions**

Who is working on what?

daemon
integration-cli
pkg
registry
container
volume
internal
hack
opts
profiles
distribution
reference
cli
plugin
cmd
contrib
layer
image
integration
api
builder

codescene.io

CodeScene | CodeScene

CODESCENE CLOUD

Plans Documentation Log in

ANALYSIS RESULTS

DOCKER

- Dashboard
- Scope
- Technical Debt
- Architecture
- Social Analyses
  - Social Networks
  - Individuals
  - Teams
  - Authors
- Project Management
  - Branches

Identify the primary developers behind the code to coordinate and support onboarding.

Owners | Knowledge Loss | Coordination Needs

System / docker

- api
- builder
- cli
- client
- cmd
- container
- contrib
- daemon
- distribution
- dockerversion
- errdefs
- hack
- image
- integration
- integration-cli
- internal
- layer
- libcontainerd

● Michael Crosby
● Brian Goff
● John Howard (VM)
● Vincent Demeester
● Daniel Nephin
● Tõnis Tiigi
● David Calavera
● Aaron Lehmann
● Yong Tang
● Derek McGowan
● Tibor Vass
● Akihiro Suda
● Kenfe-Mickaël Laventure
● Alexander Larsson
● Ahmet Alp Balkan
● Allen Sun
● Josh Hawn
● Sebastiaan van Stijn
● Evan Hazlett
● Guillaume J. Charmes

About FAQ Blog Showcase Pricing Support Copyright © 2015-2019 Empear

codescene.io
CodeScene
CODESCENE CLOUD
Plans
Documentation
Log in
ANALYSIS RESULTS
DOCKER
Dashboard
Scope
Technical Debt
Architecture
Social Analyses
Social Networks
Individuals
Teams
Authors
Project Management
Branches
Owners
Knowledge Loss
Coordination Needs
client.go
request.go
container_copy_test.go
client_test.go
image_pull_test.go
config_inspect_test.go
errors.go
container_list_test.go
service_create_test.go
plugin_install.go
task_inspect_test.go
volume_list_test.go
plugin_list_test.go
container_stop_test.go
image_search.go
image_push_test.go
image_remove_test.go
image_create.go
image_history_test.go
service_inspect.go
interface.go
secret_create_test.go
utils.go
image_load.go
network_list_test.go
container_diff_test.go
build_prune.go
config_create_test.go
node_inspect.go
secret_inspect.go
service_list_test.go
secret_list_test.go
network_list.go
plugin_enable.go
config_update_test.go
hijack.go
volume_create_test.go
plugin_disable.go
container_top.go
secret_inspect_test.go
container_unpause_test.go
distribution_inspect_test.go
config_list_test.go
checkpoint_create.go
version.go
image_search_test.go
container_exec.go
swarm_update_test.go
transport.go
disk_usage.go
network_prune_test.go
image_list.go
plugin_disable_test.go
swarm_init.go
events_test.go
swarm_init_test.go
plugin_create.go
plugin_remove_test.go
network_inspect_test.go
volume_remove_test.go
task_list_test.go
image_import_test.go
image_create_test.go
task_logs.go
image_save_test.go
image_list_test.go
container_create_test.go
image_inspect_test.go
info_test.go
swarm_inspect_test.go
container_exec_test.go
image_prune_test.go
service_update.go
volume_inspect_test.go
container_logs_test.go
container_prune_test.go
image_tag_test.go
image_build.go
container_inspect_test.go
View Code
Authors
About
FAQ
Blog
Showcase
Pricing
Support

Identify the primary developers behind the code to coordinate and support onboarding.

System / docker / client / interface.go

| | |
|---|---|
| Size | 167 Lines of Code |
| Primary Author | Michael Crosby (55 %) ● |
| Knowledge Loss | 0 % Abandoned Code |
| Fragmentation | 0.68 (0.0 -> 1.0) |
| Number of Authors | 23 |

## Actions

● Michael Crosby
● Brian Goff
● John Howard (VM)
● Vincent Demeester
● Daniel Nephin
● Tõnis Tiigi
● David Calavera
● Aaron Lehmann
● Yong Tang
● Derek McGowan
● Tibor Vass
● Akihiro Suda
● Kenfe-Mickaël Laventure
● Alexander Larsson
● Ahmet Alp Balkan
● Allen Sun
● Josh Hawn
● Sebastiaan van Stijn
● Evan Hazlett
● Guillaume J. Charmes

Copyright © 2015-2019 Empear

Maybe you don't want to leverage something like CodeScene.

Once again we can turn to our SCM tool.

github.com
Why GitHub?
Enterprise
Explore
Marketplace
Pricing
Search
Sign in
Sign up

spring-projects / spring-framework

Watch 3,366 | Star 26,745 | Fork 17,073

Code | Issues 708 | Pull requests 183 | Projects 0 | Wiki | Insights

Branch: master | Create new file | Find file | History

spring-framework / spring-core / src / main / java / org / springframework / core / annotation /

jhoeller Polishing — Latest commit 106a757 12 days ago

..

| | | |
|---|---|---|
| AbstractAliasAwareAnnotationAttri... | Fix overridden methods nullability | 2 years ago |
| AliasFor.java | Polishing | 4 years ago |
| AnnotatedElementUtils.java | Clean up warning in AnnotatedElementUtils | 6 months ago |
| AnnotationAttributeExtractor.java | Consistent use of @nullable across the codebase (even for internals) | 2 years ago |
| AnnotationAttributes.java | Polishing | 12 days ago |
| AnnotationAwareOrderComparator.... | Correctly delegate to OrderUtils.getPriority for DecoratingProxy | 10 months ago |
| AnnotationConfigurationException.... | Exception fine-tuning and general polishing | 4 years ago |
| AnnotationUtils.java | Consistently skip unnecessary search on superclasses and empty elements | 6 months ago |
| DefaultAnnotationAttributeExtracto... | Fix overridden methods nullability | 2 years ago |
| MapAnnotationAttributeExtractor.ja... | Fix overridden methods nullability | 2 years ago |
| Order.java | Polishing | a year ago |
| OrderUtils.java | Pruning of outdated JDK 6/7 references (plus related polishing) | 7 months ago |
| SynthesizedAnnotation.java | Make SynthetizedAnnotation public | 4 years ago |
| SynthesizedAnnotationInvocationH... | Consistent alias processing behind AnnotatedTypeMetadata abstraction ... | 3 years ago |
| SynthesizingMethodParameter.java | MethodParameter supports Java 8 Executable/Parameter and validates pa... | 3 years ago |
| package-info.java | Ensure all files end with a newline | 8 months ago |

Last commit around the Super Blue Blood Moon Eclipse?

Probably not a great candidate for microservices then!

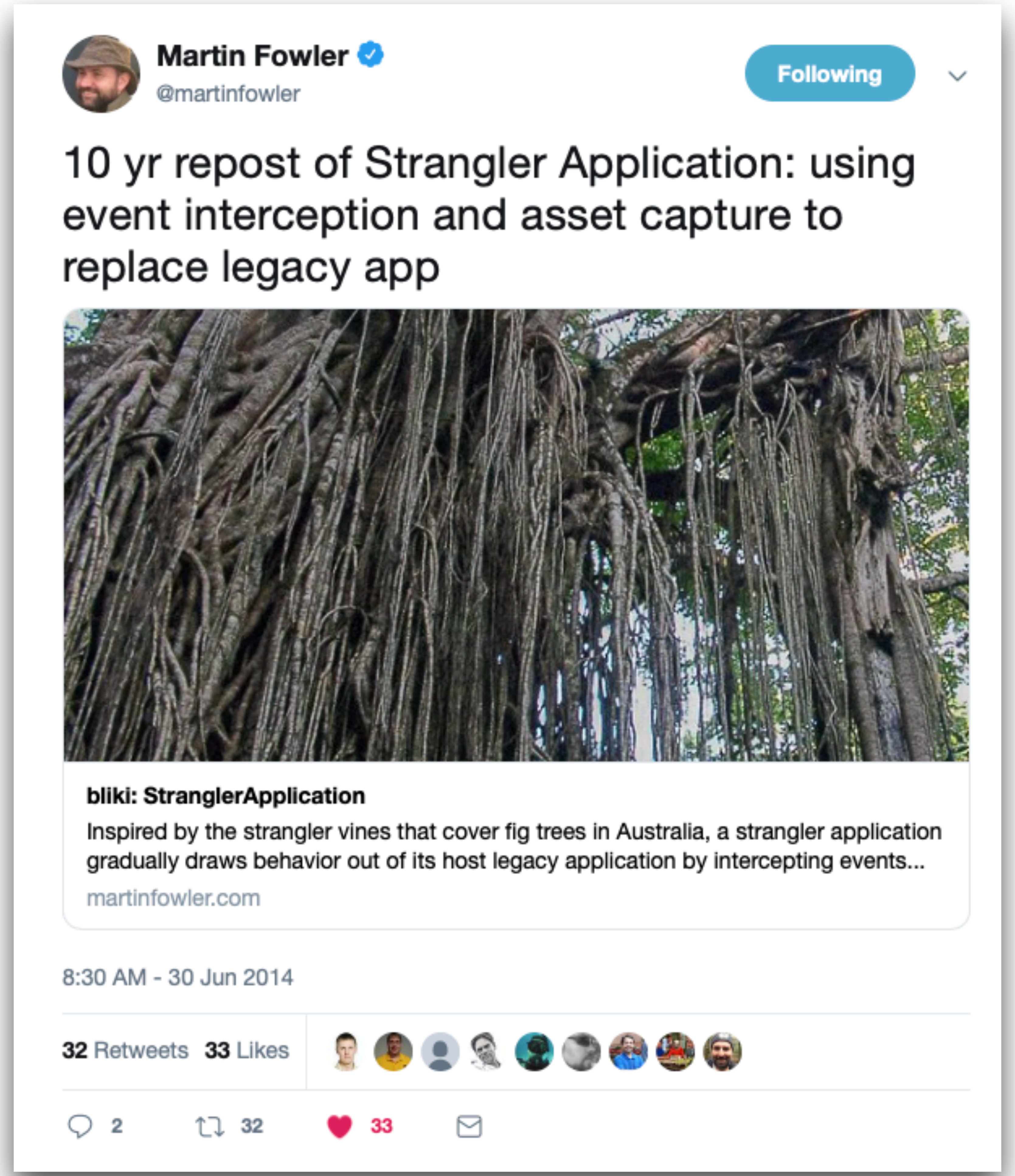
Find a spot that “always be changing”? Dig deeper!

Look at your bug tracker -
look for defect density.

Look at your backlog. Where is the locus of attention?

We have some good candidates…now what?

The Strangler Pattern to the rescue.



Martin Fowler
@martinfowler

Following

10 yr repost of Strangler Application: using event interception and asset capture to replace legacy app

**bliki: StranglerApplication**
Inspired by the strangler vines that cover fig trees in Australia, a strangler application gradually draws behavior out of its host legacy application by intercepting events...
martinfowler.com

8:30 AM - 30 Jun 2014

32 Retweets 33 Likes

2 32 33

https://twitter.com/martinfowler/status/483603425008304129

Strangler Application.

https://martinfowler.com/bliki/StranglerApplication.html

In a nutshell, build the new around the edges of the old.

Gradually replace
the heritage bits.

Reduces the risk of a
big bang cutover.

Incrementally improve delivering business value as you go.

Able to show regular progress to stakeholders.

We can go a step further and apply a data driven approach.

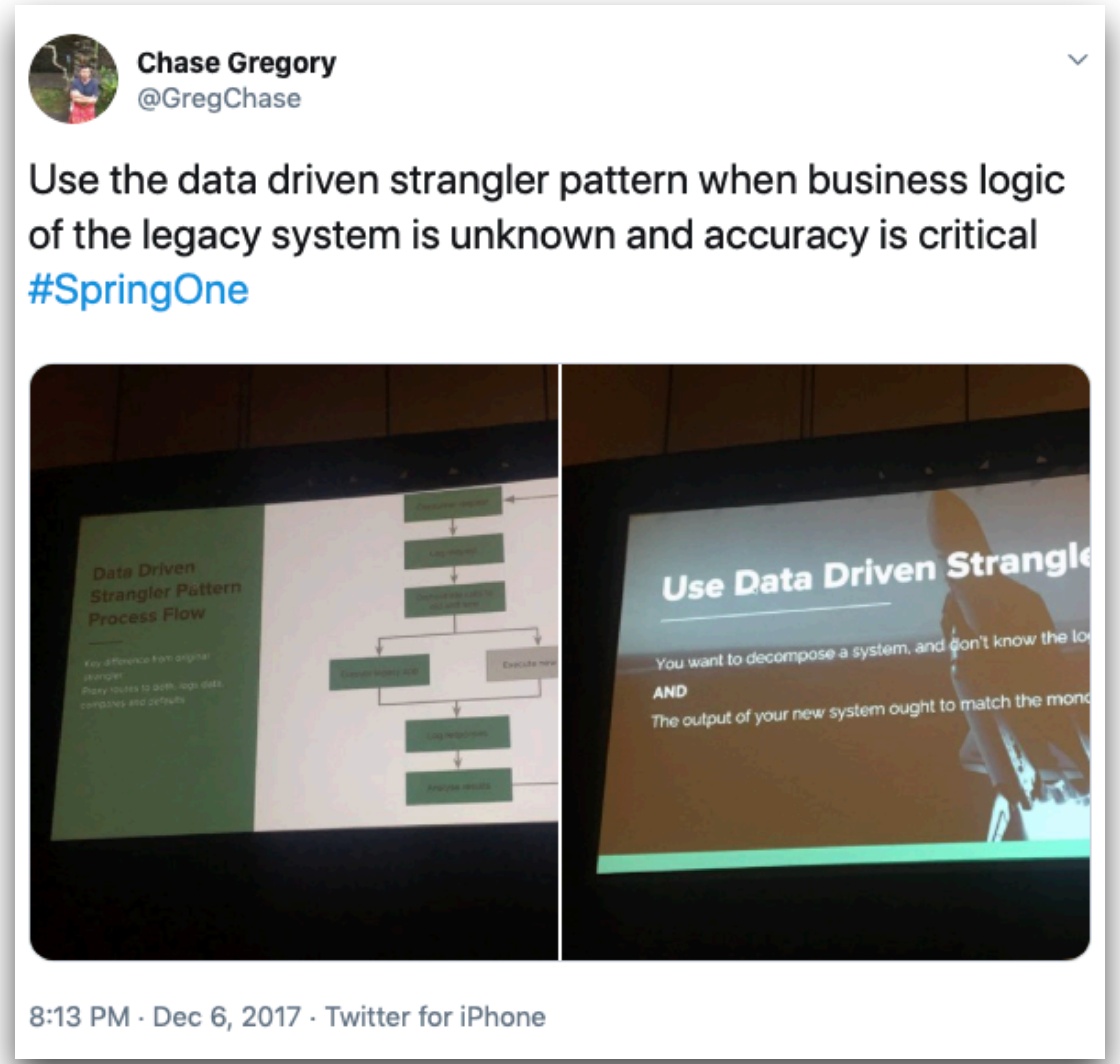
What would you say the old system does exactly?

¯\_(ツ)_/¯

Odds are we don’t understand all the nuance of the old bits.

Leads to bugs, no one knew about that edge case…

## What if we had real world data?



Chase Gregory
@GregChase

Use the data driven strangler pattern when business logic of the legacy system is unknown and accuracy is critical #SpringOne

Data Driven Strangler Pattern Process Flow
Use Data Driven Strangle
You want to decompose a system, and don't know the lo
AND
The output of your new system ought to match the mon

8:13 PM · Dec 6, 2017 · Twitter for iPhone

https://mobile.twitter.com/GregChase/status/938592224924725248

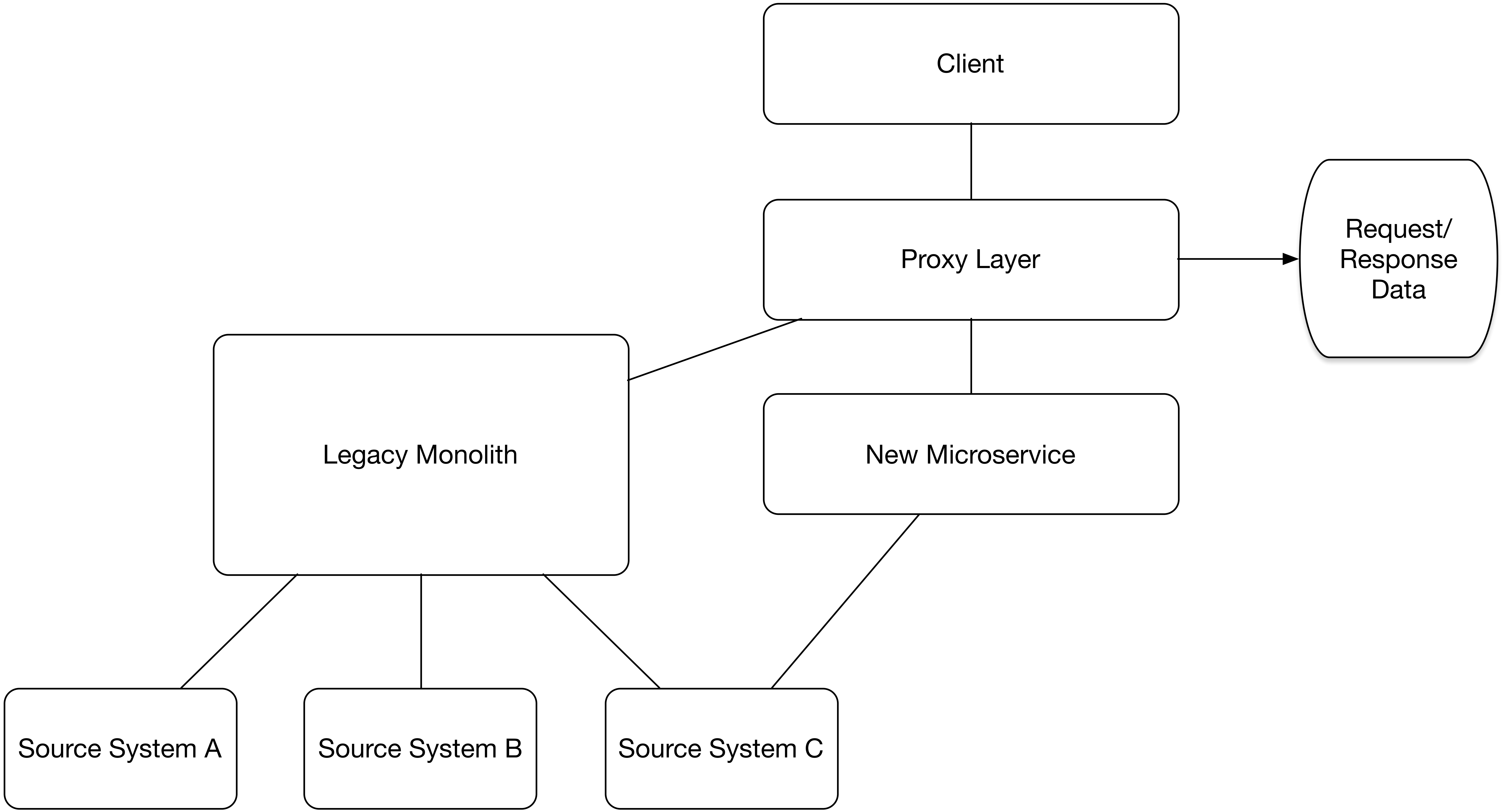
Put a proxy layer between the client and the legacy system.

https://content.pivotal.io/slides/strangling-the-monolith-with-a-data-driven-approach-a-case-study

Log the results -
requests and responses.

You now know what the old system does.

Drives test cases for the new functionality.


Client
Proxy Layer
Request/ Response Data
Legacy Monolith
New Microservice
Source System A
Source System B
Source System C

You can run the new in parallel with the old.

Route requests to both modules - compare the results.

If they match, #winning.

In they don’t, you can use the “heritage” response…

And then add tests to figure out which result is correct.

Don’t be surprised if the old system is wrong!

# INDEPENDENT LIFE CYCLES

Monoliths are big ships - they don’t turn on a dime.

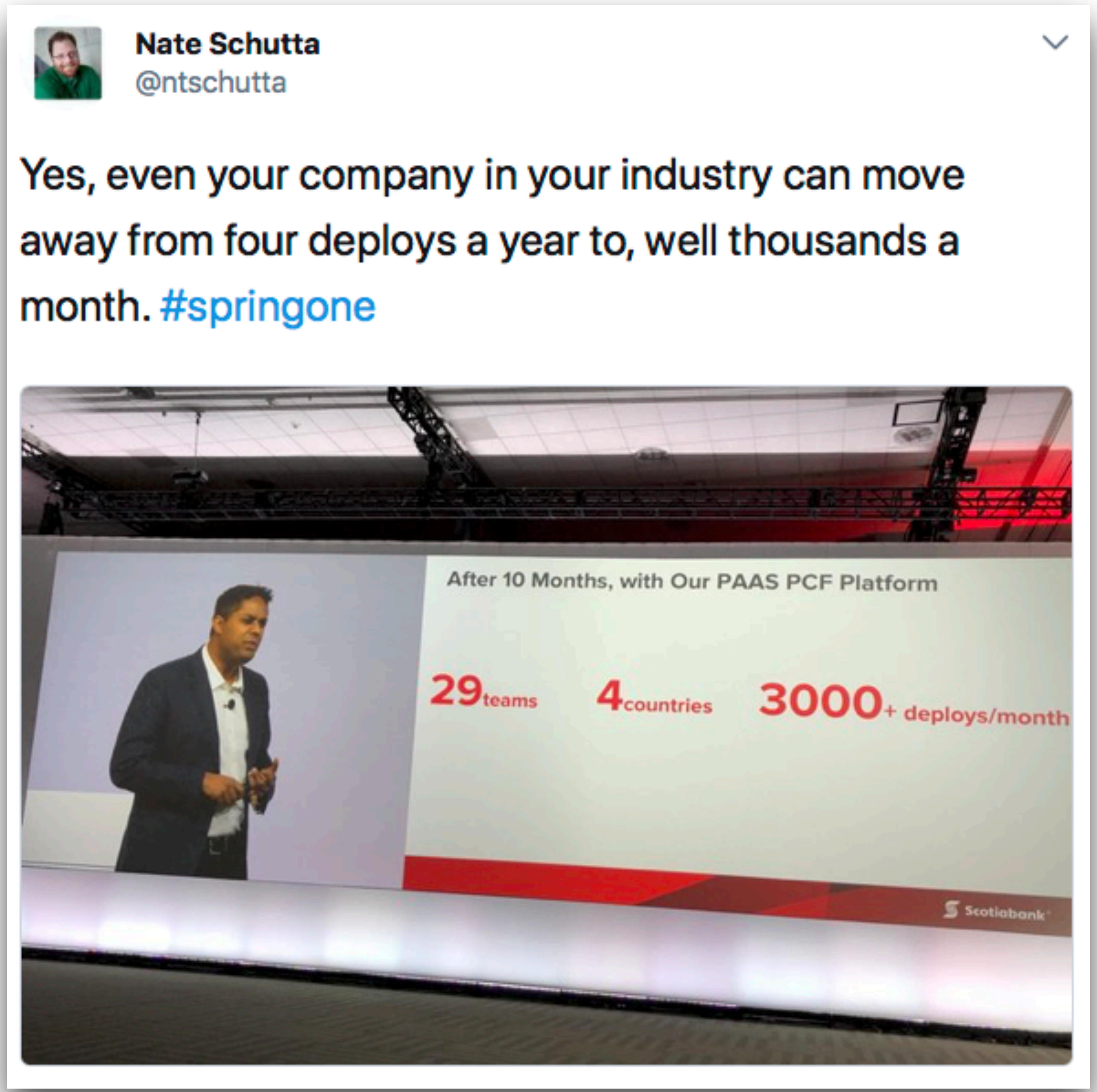
But that doesn‘t work today.

Always be changing.

Run experiments. A/B testing.

Respond to business changes.

Deliver in days not months.


Nate Schutta
@ntschutta

Yes, even your company in your industry can move away from four deploys a year to, well thousands a month. #springone

https://mobile.twitter.com/ntschutta/status/938109379995353088

Speed matters.

Disruption impacts every business.

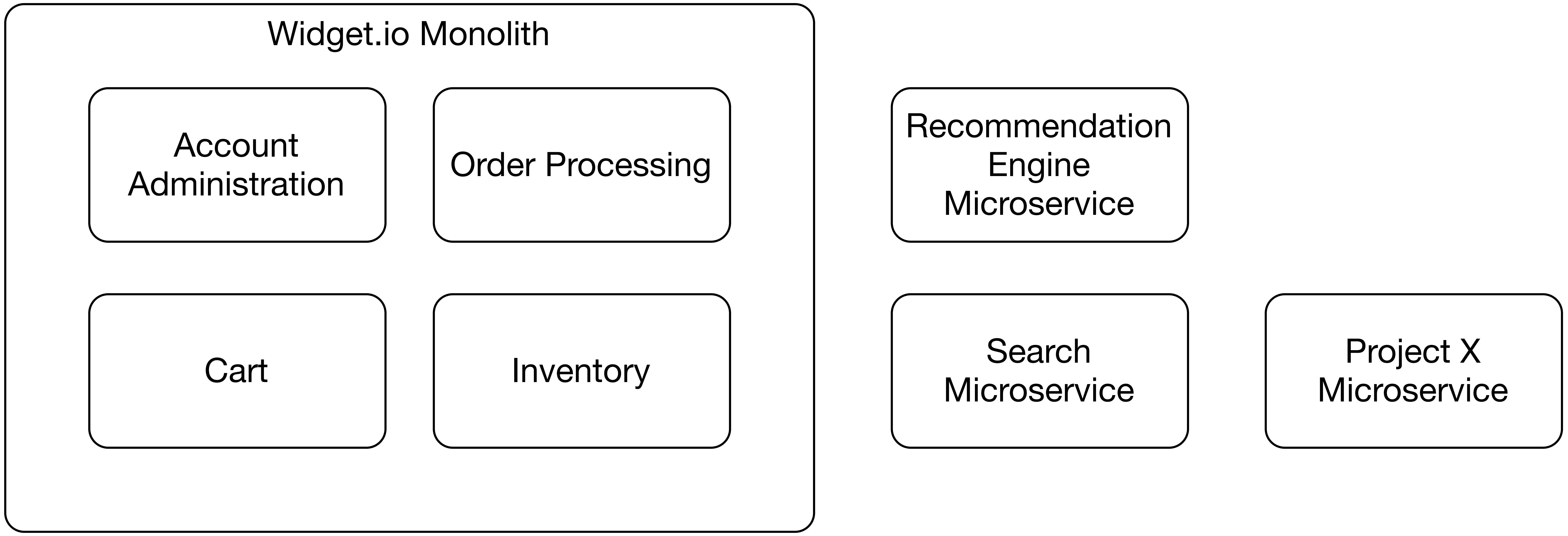
Your industry is not immune.

## Returning to our Widget.io Monolith…

What if our business identify a new opportunity…

But it requires us to iterate
and deliver in days.

The quarterly release cycle won’t cut it. What do we do?


Widget.io Monolith
Account Administration
Order Processing
Cart
Inventory
Recommendation Engine Microservice
Search Microservice
Project X Microservice

As a microservice, Project X is independent of the rest.

It has its own repository
and build pipeline.

In other words it has an independent life cycle.

But we don't *just* get speed to market.

Increases developer productivity!

Monoliths often have dictionary sized getting started guides.

Build times measured in phases of the moon.

It can take months for a new developer to get up to speed.

What was your longest stretch to get to productive team member?

Smaller scope === less to get your head wrapped around.

Builds take a minute or two.

Build breaks are fixed promptly.

Testing can be far simpler.

Goodbye 80 hour manual regression suites.

Hello fine grained tests
that run on every commit.

Forget the one-off performance test.

Use the right tools for the job!

Shared life cycles put us at the mercy of the longest tent pole.

We are no longer forced into a one size fits none approach.

Each microservice can use the mix of tests that make sense.

Use the appropriate linting rules and code quality scans.

Simplifies the search for fitness functions.

https://www.thoughtworks.com/insights/blog/microservices-evolutionary-architecture

We can practice hypothesis driven development.

https://www.thoughtworks.com/insights/blog/how-implement-hypothesis-driven-development

“Prediction is very difficult, especially if it's about the future.”

–Niels Bohr (attributed)

## Ever debate possible solutions?

“My approach will clearly increase conversions.”

How do you know?

## What happens if you’re wrong?

In the monolith, we had to be conservative.

Now - we can test our hypothesis.

A/B test it!

We believe <this change>
Will result in <this outcome>
We will know we have succeeded
when <we see a X change in
this metric>

We believe adding a distributed cache
Will result in faster startup times
We will know we have succeeded if startup time is less than 15 seconds

Can lead to useful fitness functions.

A/B used to be limited to tech giants like Amazon and Google.

Now it is within reach for all of us!

What customer doesn't want a constantly improving product?

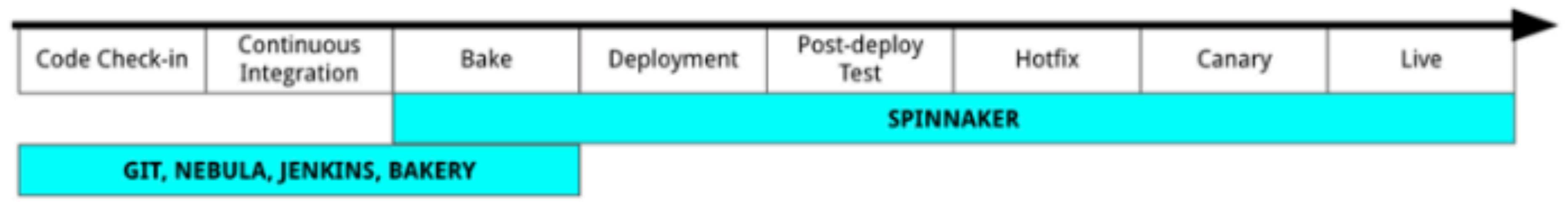
Of course for this to work…

Focus on “paved roads.”

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

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.

Expertise grows with repetition.

Deploy early, deploy often.

You will improve.

Need to develop trust
in the process.

We need robust pipelines.

Concourse, Visual Studio Team Services, and Jenkins can help.

Not sure how to create a pipeline?

Spring Cloud Pipelines.

https://spring.io/blog/2018/11/13/spring-cloud-pipelines-to-cloud-pipelines-migration

Opinionated build/
test/stage/prod flow.

Gives you a place to start -
modify to your hearts content.

Independent life cycles very under appreciated benefit.

“That’s how we’ve always done it” won’t cut it anymore.

## SCALE INDEPENDENTLY

The monolith forced us to make decisions early.

Often when we knew the least.

For example - how much capacity will you need?

¯_(ツ)_/¯

Take worst case…double it…add some buffer. Then a bit more.

Just in case.

We have a six week (aka month) lead time on all requests.

Lots of tickets.

And meetings.

And email.

And followup.

It was in our best interest to over allocate resources.

Better to have it and not need it…

Difficult to add more capacity later.

Gave us single digit resource utilization.

Of course not all traffic is predictable is it?

Matters were much worse if we had unexpected demand.

We can plan for a big new initiative.

But a shout out on social media might double our traffic in minutes.

Things were no easier for our operations staff.

Annual budgets make it difficult to add capacity smoothly.

Cloud environments and microservices flip the script.

Today we can add, and remove, capacity on demand.

We can wait for the last responsible moment.

Instead of swags and guesses.

Not surprisingly, the monolith suffered from the same issue.

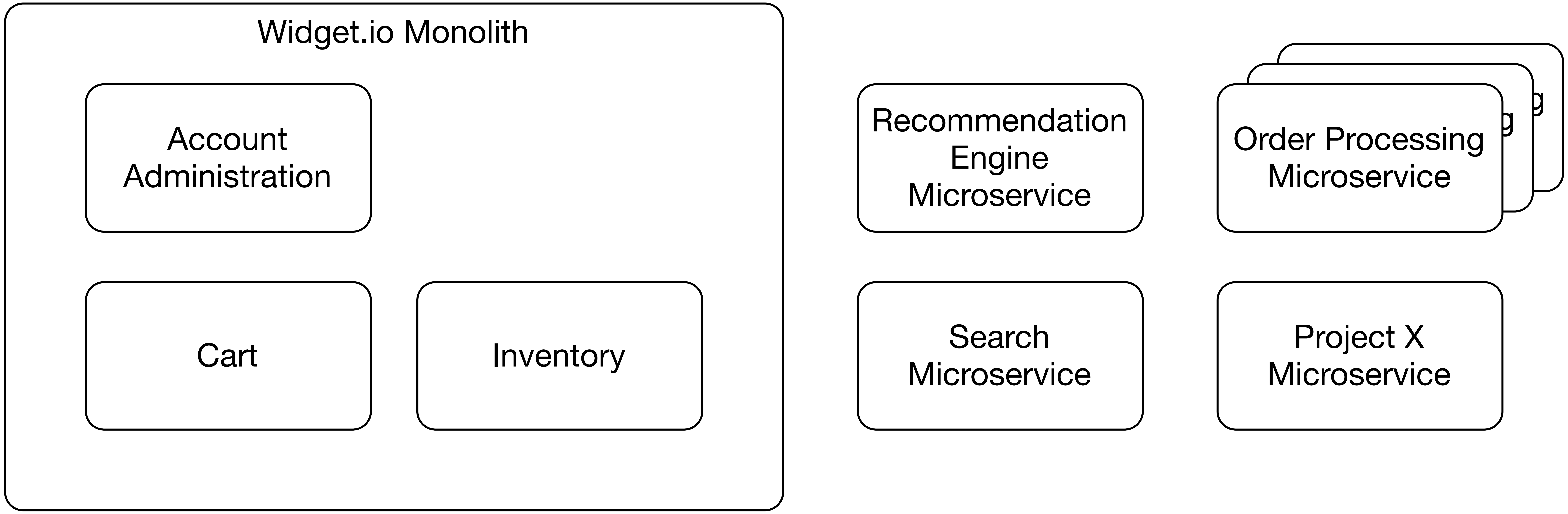
There was no way to scale “just the parts that needed it”.

It was all or nothing.

Which again, meant we were often heavily over allocated.

Harkening back to the Widget.io example…

Odds are the order processing system has a unique scaling needs.

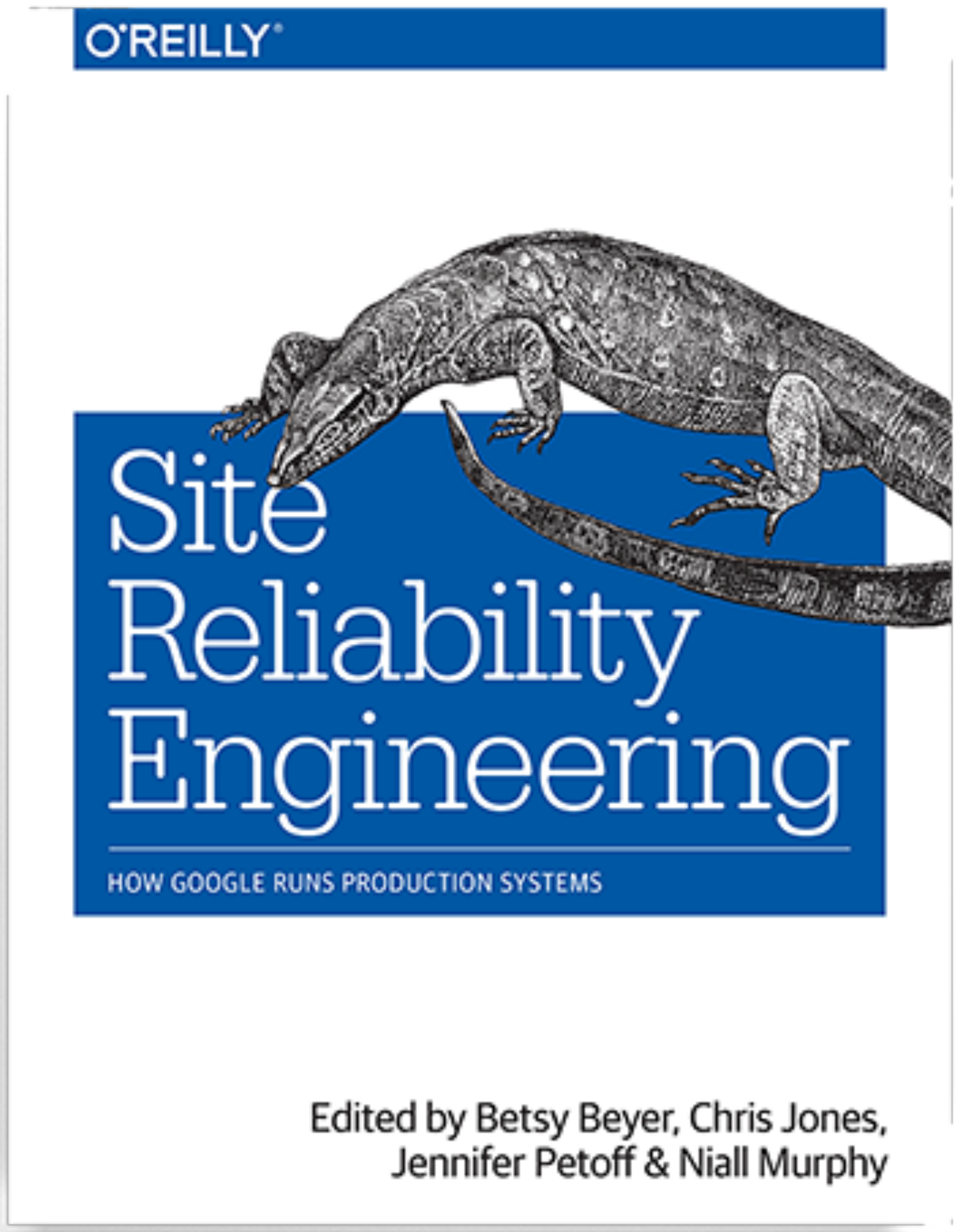

Widget.io Monolith
Account Administration
Cart
Inventory
Recommendation Engine Microservice
Order Processing Microservice
Search Microservice
Project X Microservice

With a microservices approach,
we can fully utilize compute.

But, how do we know which components need more capacity?

## Monitoring to the rescue!

Monitoring is vital to a thriving microservices architecture.


O'REILLY
Site Reliability Engineering
HOW GOOGLE RUNS PRODUCTION SYSTEMS
Edited by Betsy Beyer, Chris Jones, Jennifer Petoff & Niall Murphy

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

## Four Golden Signals.

https://landing.google.com/sre/book/chapters/monitoring-distributed-systems.html#xref_monitoring_golden-signals

Latency - how long does it take to service a request.

Traffic - level of demand on the system. Requests/second. I/O rate.

Errors - failed requests. Can be explicit, implicit or policy failure.

Saturation - how much of a constrained resource is left.

Important to consider the sampling frequency.

High resolution can be costly.

## Aggregate data.

Number of tools from PCF to Dynatrace to New Relic.

## Spring Boot Actuator!

https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-metrics.html

## 57. Metrics

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer, an application metrics facade that supports numerous monitoring systems, including:

- AppOptics
- Atlas
- Datadog
- Dynatrace
- Elastic
- Ganglia
- Graphite
- Humio
- Influx
- JMX
- KairosDB
- New Relic
- Prometheus
- SignalFx
- Simple (in-memory)
- StatsD
- Wavefront

To learn more about Micrometer's capabilities, please refer to its reference documentation, in particular the concepts section.

### 57.1 Getting started

Spring Boot auto-configures a composite `MeterRegistry` and adds a registry to the composite for each of the supported implementations that it finds on the classpath. Having a dependency on `micrometer-registry-{system}` in your runtime classpath is enough for Spring Boot to configure the registry.

Most registries share common features. For instance, you can disable a particular registry even if the Micrometer registry implementation is on the classpath. For instance, to disable Datadog:

```
management.metrics.export.datadog.enabled=false
```

Takes time to get monitoring right.

## Do you even SRE?

Beware the metric that is easy to measure…

Might not be meaningful. Sorry.

Also key to understand the business drivers.

## What could cause a spike in demand?

## How does that translate to specific services?

Be realistic!

We can’t all be a third
of internet traffic!

Independent scalability is a massive win. If you need it!

# FAILURE ISOLATION

No service is an island.

"You've taken your first step into a larger world."

–Obi-Wan Kenobi

https://www.youtube.com/watch?v=535Zy_rf4NU

No microservice works alone.

Name implies as much!

Integrations are as old as software.

Often use bailing twine and duct tape…

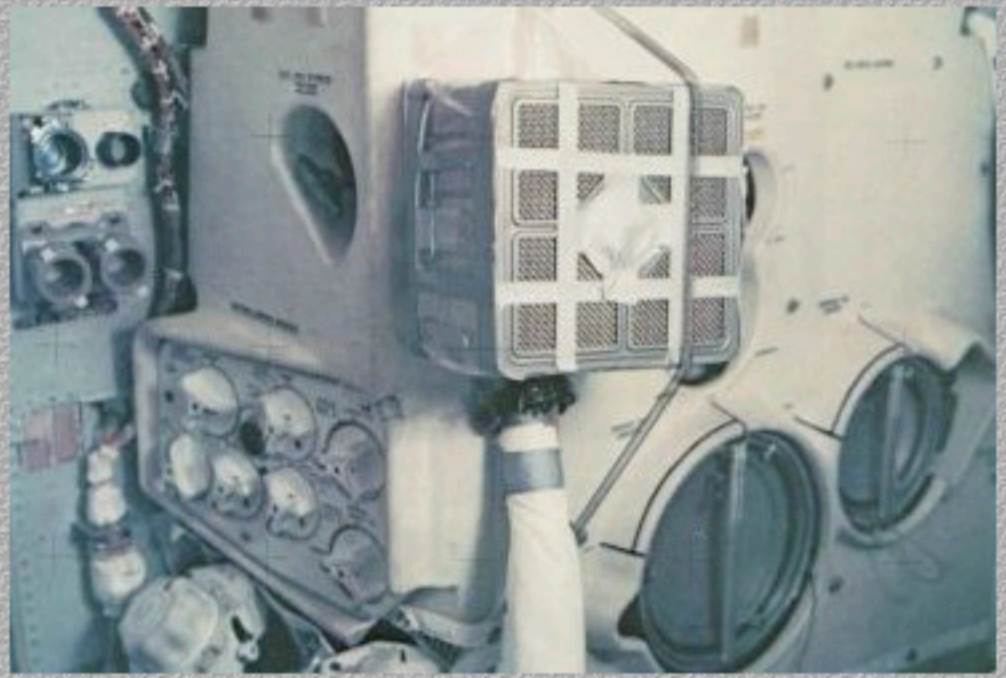
Apollo Expeditions to the Moon

# CHAPTER 13.4

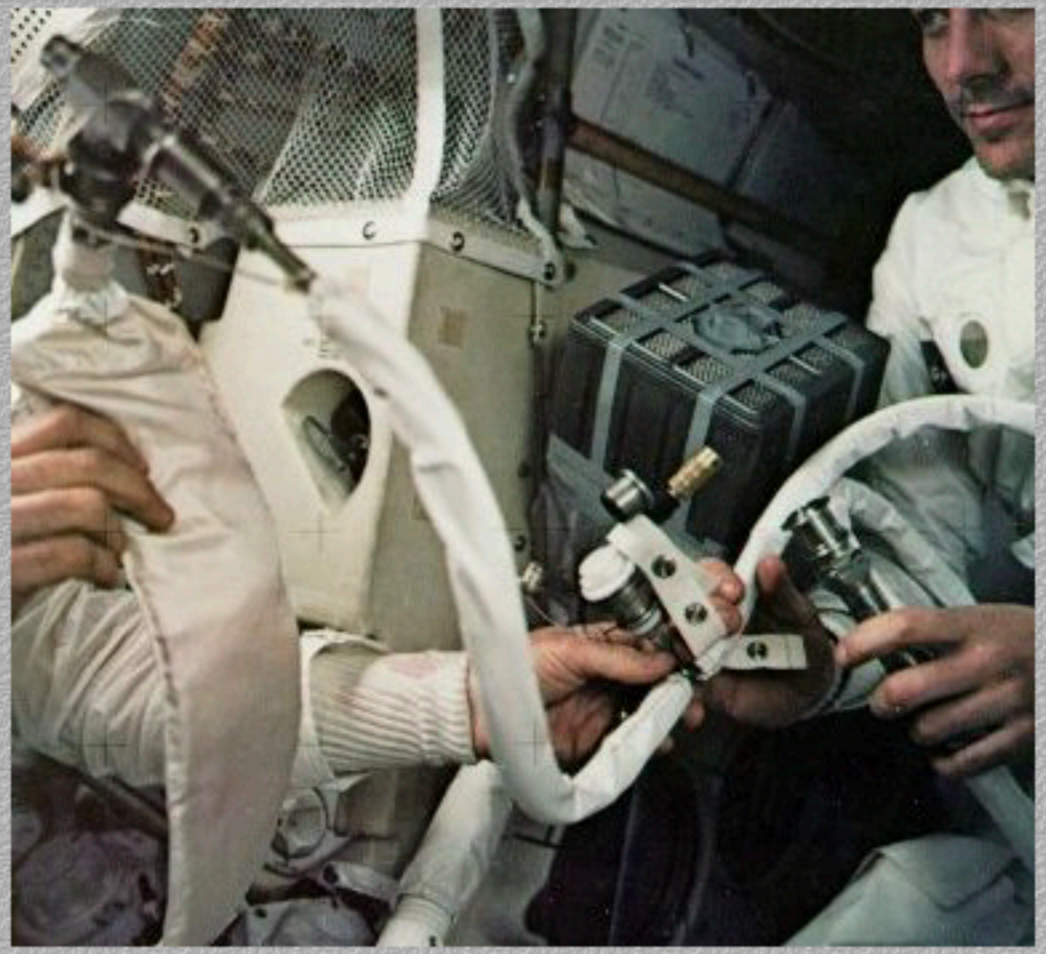
## A SQUARE PEG IN A ROUND HOLE

We would have died of the exhaust from our own lungs if Mission Control hadn't come up with a marvelous fix. The trouble was the square lithium hydroxide canisters from the CM would not fit the round openings of those in the LM environmental system. After a day and a half in the LM a warning light showed us that the carbon dioxide had built up to a dangerous level, but the ground was ready. They had thought up a way to attach a CM canister to the LM system by using plastic bags, cardboard, and tape- all materials we had on board. Jack and I put it together: just like building a model airplane. The contraption wasn't very handsome, but it worked. It was a great improvisation- and a fine example of cooperation between ground and space.

The big question was, "How do we get back safely to Earth?" The LM navigation system wasn't designed to help us in this situation. Before the explosion, at 30 hours and 40 minutes, we had made the normal midcourse correction, which would take us out of a free-return-to-Earth trajectory and put us on our lunar landing course. Now we had to get back on that free-return course. The ground-computed 35-second burn, by an engine designed to land us on the Moon, accomplished that objective 5 hours after the explosion.



"Backroom" experts at Mission Control worked many hours to devise the fix that possibly kept the astronauts from dying of carbon dioxide. CapCom Joe Kerwin led Astronaut Swigert, step by step, for an hour to build a contraption like the one the experts had constructed on Earth. It involved stripping the hose from a lunar suit and rigging the hose to the taped-over CM double canister, using the suit's fan to draw carbon dioxide from the cabin through the canister and expel it back into the LM as pure oxygen.



Sometimes those 3rd party dependencies don’t meet our SLO.

They fail.

Failures, uh find a way.

Our customers don’t care why.

We can use microservices to isolate those failure cases!

You might already know where the problem code lives.

But don't be afraid to perform an architectural review.

Look for failure points.

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.

“When month end falls on the Super Blue Blood Moon.”

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

But we need to think about the road less traveled.

What systems does our service talk to? How do they integrate?

Is it a direct call? Through a proxy?

## What are the SLOs?

Do we all have a shared understanding of what the app?

There will be gaps in knowledge.

Feature not a bug.

We now understand the failure cases, what do we do about it?

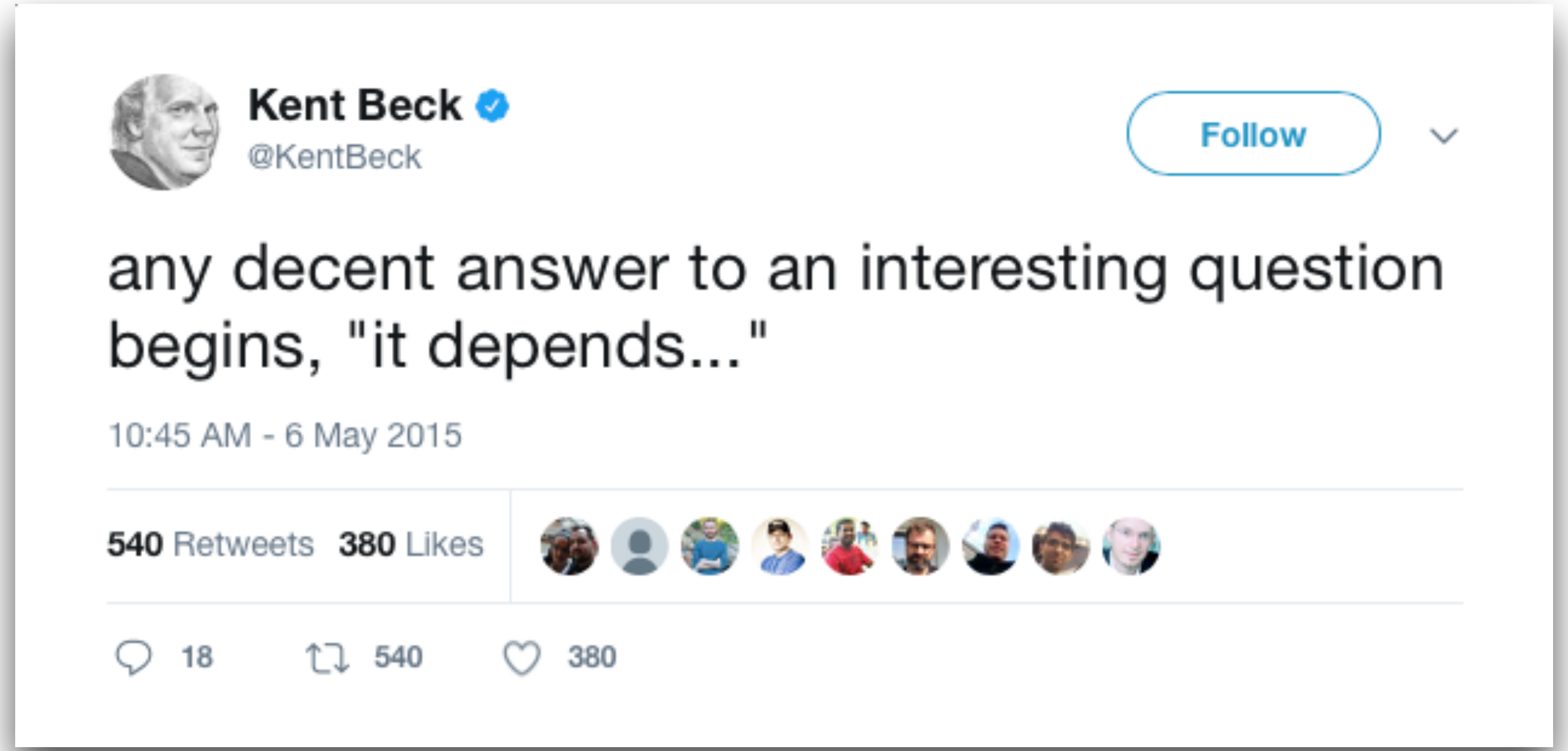
## How should we react?

Error message?

Call a backup service?

## Do we need to cache data?

Do we return a default answer?



Kent Beck
@KentBeck

any decent answer to an interesting question begins, "it depends..."

10:45 AM - 6 May 2015

540 Retweets 380 Likes

18 540 380

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

## The circuit breaker pattern.

https://martinfowler.com/bliki/CircuitBreaker.html

Closed
on call / pass through
call succeeds / reset count
call fails / count failure
threshold reached / trip breaker
trip breaker
Open
on call / fail
on timeout / attempt reset
trip breaker
reset
Half-Open
on call / pass through
call succeeds / reset
call fails / trip breaker

Circuit breaker watches the calls.

Once they exceed a failure threshold, the circuit is opened.

Redirects to the
fallback mechanism.

Periodically checks to see if the service is repaired.

If so, circuit is closed.

You won’t think of everything.

NETFLIX

https://github.com/Netflix/SimianArmy

Chaos engineering.

http://principlesofchaos.org

# INDIRECTION LAYER

As an architect there is one pattern I use often.

Another layer of indirection.

Sometimes it is overkill.

When was the last time you swapped out your database?

OK, it happens…

Same basic concept as failure isolation. With a twist.

Now we protect our service from things that change.

Or things that are complex to use.

Could be a vendor dependency.

Could be something large like an ERP system.

Or maybe just a library for currency conversion.

An indirection layer isolates the things we need to change.

If we have to swap something out,
we don’t update every client.

## Basic proxy pattern.

Can also be an instance of an adaptor.

Make this US plug fit into
an EU outlet for example.

We can also use it to simplify the interaction.

Many 3rd party dependencies solve a lot of problems.

Many of which may
not matter to us.

Our microservice can facade that interaction. Simplify it.

Nothing new here - classic Gang of Four pattern!

These facades can also supply context.

Maybe a payment gateway needs your CHQ address.

Or you need an authorization token.

That won’t change call to call.

Don’t want to code it into every client.

The facade is a natural spot for such functionality.

Maybe we want to inject some behavior before or after calls.

A indirection layer provides a natural extension point.

Architecture is often defined as the decisions that are hard to change.

Or the decisions we wish we got right.

But we *know* things will change!

Isn’t this approach anti agile?

Contributes to the “we’re agile, we don’t have architects” theory.

You definitely have people making architectural decisions!

Sure hope they are making good ones…

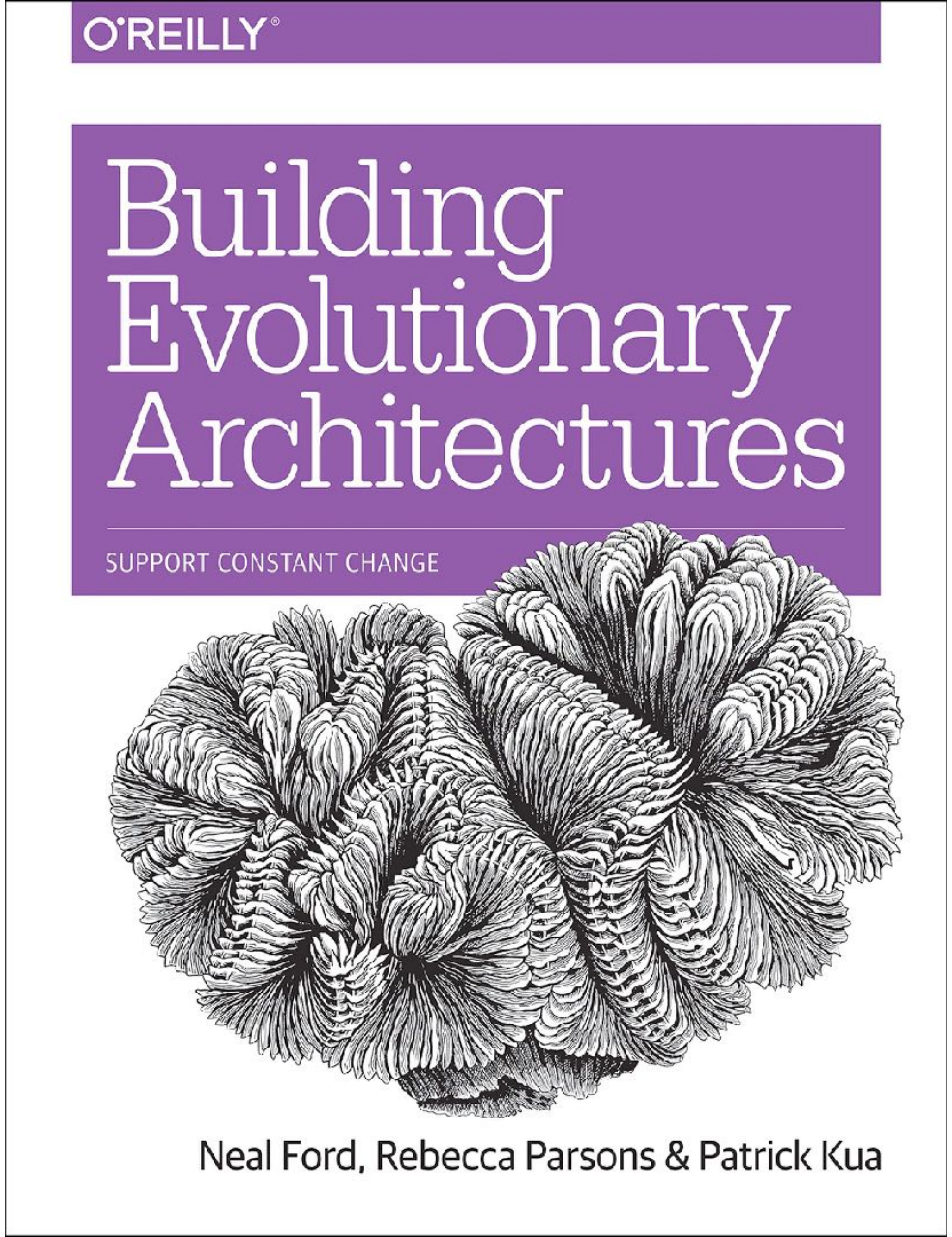
You’ll know in a year or two.

“Our app has 4 different UI frameworks…”

## What do we do about that?

Maybe we should
change our assumptions.

What if our architectures expected to change?


O'REILLY®

Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE

Neal Ford, Rebecca Parsons & Patrick Kua

http://evolutionaryarchitecture.com

“An evolutionary architecture supports guided, incremental change across multiple dimensions.”

– Building Evolutionary Architectures

Microservices can provide additional flexibility.

# POLYGLOT TECH STACKS

Monoliths forced us to standardize on a toolkit.

Many organizations described themselves by their stack.

As in “we’re a Java/Ruby/.NET shop“

Bring me a problem!

There are positives to this approach.

Teams develop deep expertise.

People can shift teams to cross pollinate and balance workloads.

Simplifies the hiring and training processes.

Ops can focus on the primary environment.

But one size doesn’t fit all.

There are, of course, downsides.

Currency is usually constrained by the slowest moving app.

You can’t have nice things because of the Wombat app.

When we did upgrade, odds are it would take months.

And the “new” version would already be outdated.

Of course very few orgs were really that homogenous.

A merger or acquisition ===
another tech stack.

Cloud computing removes the one stack to rule them all constraint.

We actually can spin up multiple different stacks.

Polyglot programming isn’t just a pipe dream anymore!

http://nealford.com/memeagora/2006/12/05/Polyglot_Programming.html

Pick the right tool for the job!

We aren’t forced down the square peg round hole path.

But.

There is always a but.

We have to avoid tech sprawl.

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

Every developer 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?

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.

nbcnews.com

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

techcrunch.com

# Most of the Fortune 100 still use flawed software that led to the Equifax breach

**Zack Whittaker**
@zackwhittaker / 1 week ago

Almost two years after Equifax's massive hack, the majority of Fortune 100 companies still aren't learning the lessons of using vulnerable software.

In the last six months of 2018, two-thirds of the Fortune 100 companies downloaded a vulnerable version of Apache Struts, the same vulnerable server software that was used by hackers to steal the personal data on close to 150 million consumers, according to data shared by Sonatype, an open-source automation firm.

That's despite almost two years' worth of patched Struts versions being released since the attack.

Sonatype wouldn't name the Fortune 100 firms that had downloaded the cnet.com

SECURITY / LEER EN ESPAÑOL

# Exactis said to have exposed 340 million records, more than Equifax breach

We hadn't heard of the firm either, but it had data on hundreds of millions of Americans and businesses and leaked it, according to Wired.

BY ABRAR AL-HEETI / JUNE 28, 2018 10:14 AM PDT

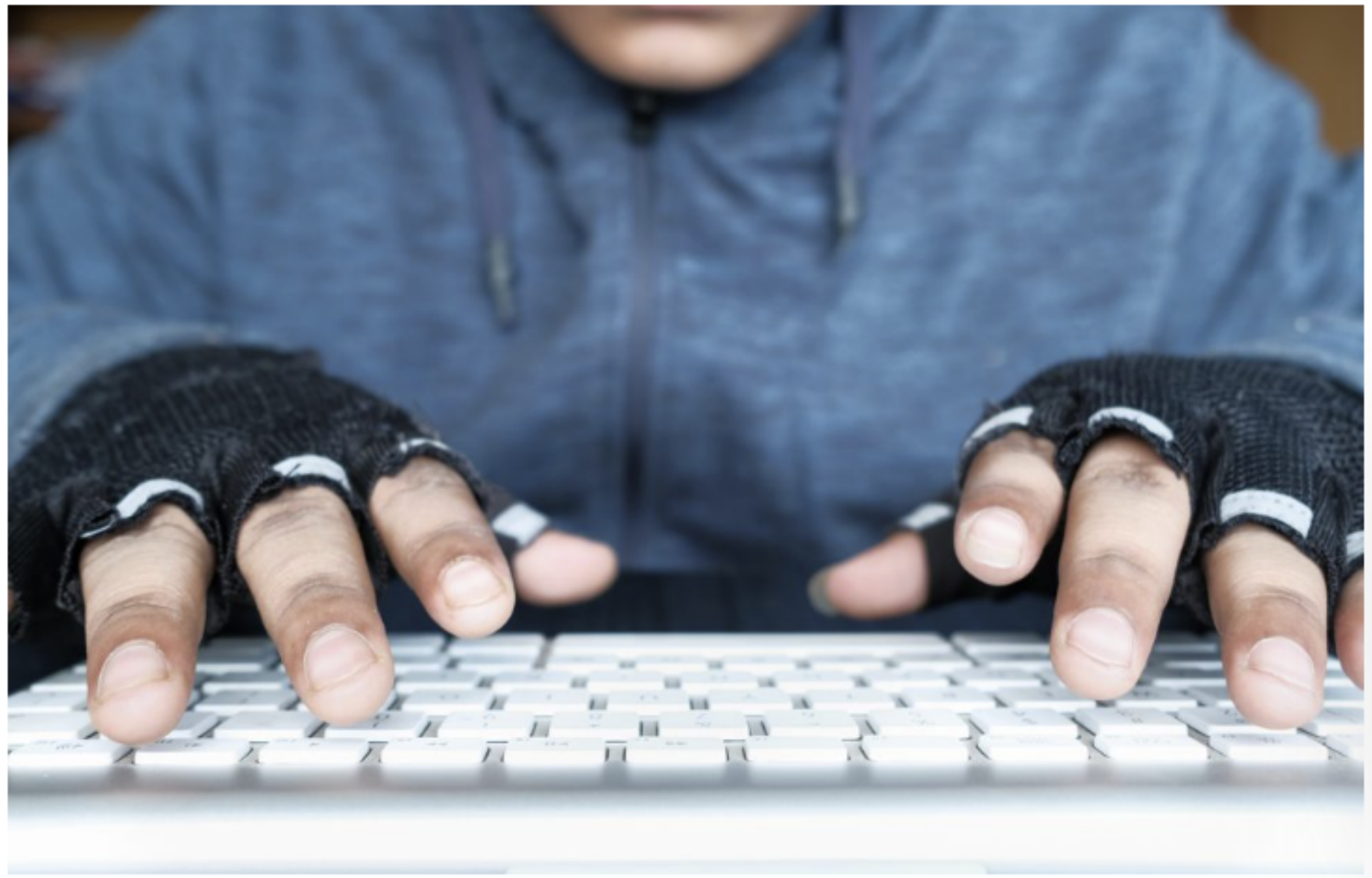
2018 MKC

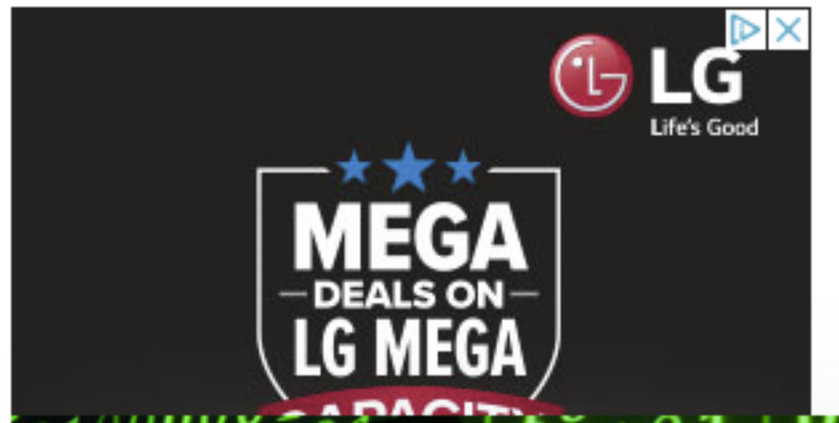
Introduce yourself to a new Lincoln.

CURRENT OFFERS

BUILD & PRICE

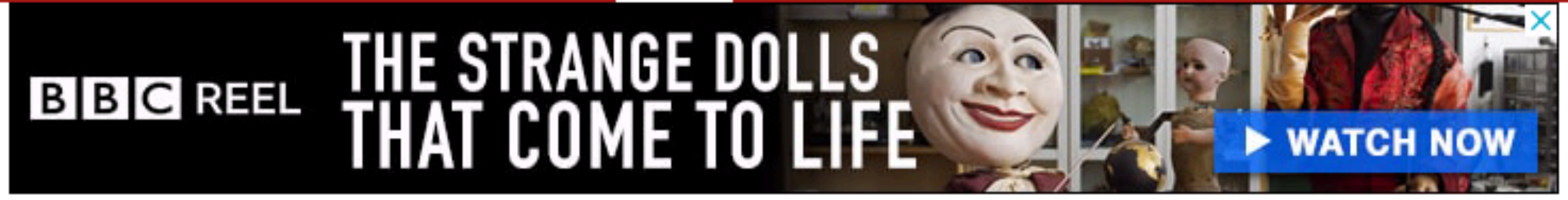
Roll over for offer disclaimer





Worst hacks of the year 00:00 / 03:24

bbc.com



Technology

# Marriott hack hits 500 million Starwood guests

30 November 2018

Share

Sheraton is one of Marriott's brands

**The records of 500 million customers of the hotel group Marriott International have been involved in a data breach.**

The hotel chain said the guest reservation database of its Starwood division had been compromised by an unauthorised party.

It said an internal investigation found an attacker had been able to access the

## Top Stories

**Tabloid's owner defends Jeff Bezos report**
AMI, owner of a US magazine accused of blackmail by Amazon's founder, says it acted in good faith.
40 minutes ago

**What US ruling may mean for Roe v Wade**
2 hours ago

**Russia probe chief grilled by lawmakers**
24 minutes ago

## Features

It cannot be a free for all.

You will need some guardrails.

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

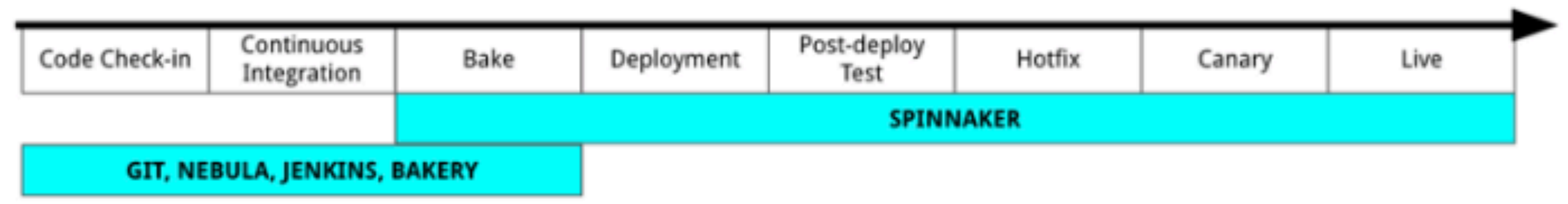
Pick from these 3 flavors. Won’t work for you? Let’s talk.

## Focus on “paved roads.”

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

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.

The key word here is micro.

As in small.

We can debate the meaning of small until the cows come home.

Partial to “anything we can rewrite in 2 weeks or less”.

If we chose poorly - we lost two weeks. An iteration.

We can recover from that.

More time === more invested.

Makes us less likely to change course. Even if we should.

## Microservices frees us to choose the right tech!

But we must weigh the pros and cons.

“With great power comes great responsibility.”

–Uncle Ben

You build it, you run it.

Avoid the temptation of resume driven design.

Microservices really do offer some impressive benefits.

But they come at a price.

Don’t pay the complexity tax unless you get something in return.

In other words, no, not everything should be a microservice!

Use them were they make sense.

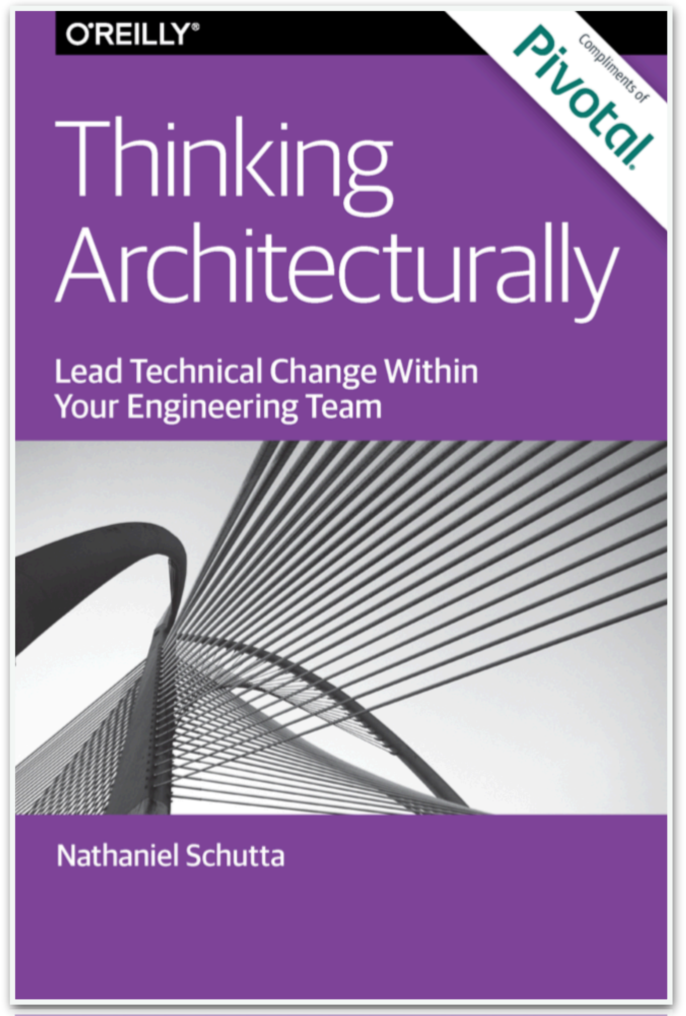
Use them where they add value.

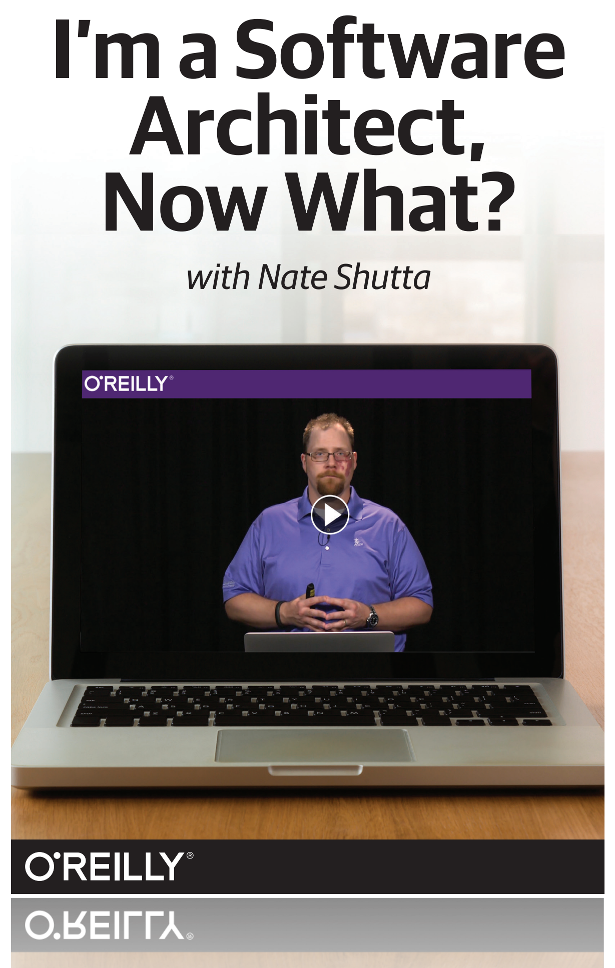
If you need one (or more) of the principles, go forth and prosper!

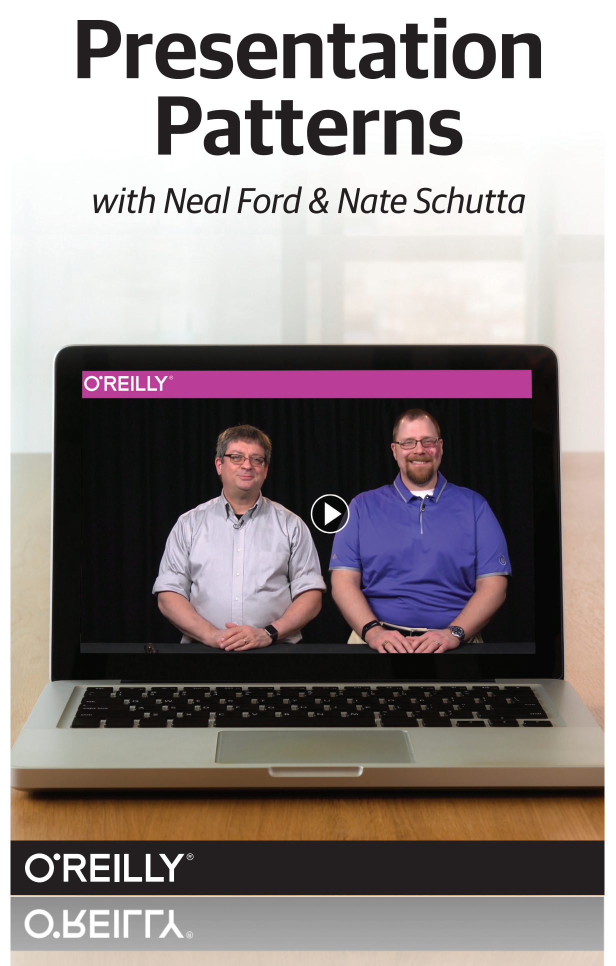
If not…well, there's always serverless.

Good luck!

# Thanks!

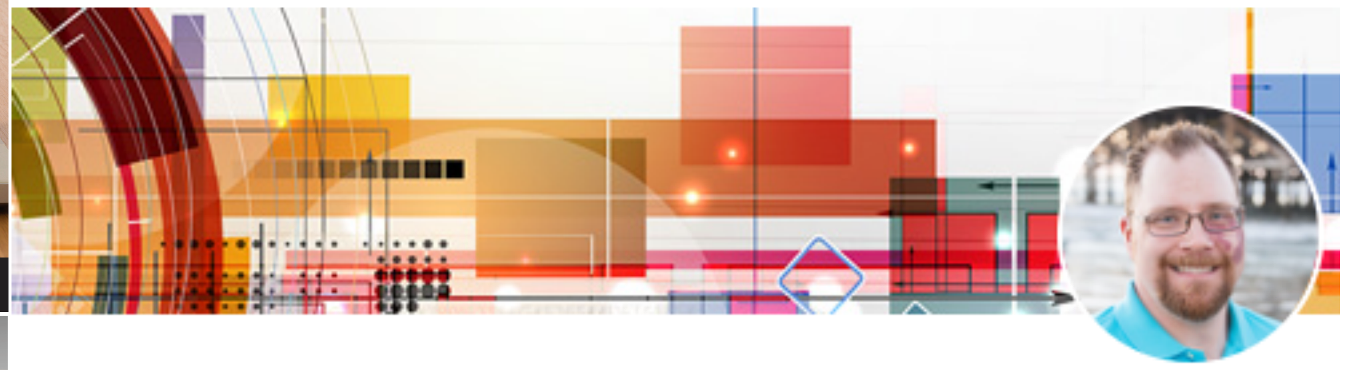

O'REILLY
Thinking Architecturally
Lead Technical Change Within Your Engineering Team
Nathaniel Schutta
Compliments of Pivotal


I'm a Software Architect, Now What?
*with Nate Shutta*
O'REILLY


Presentation Patterns
*with Neal Ford & Nate Schutta*
O'REILLY

Modeling for Software Architects
*with Nate Shutta*
O'REILLY

**Nathaniel T. Schutta**
**@ntschutta**
**ntschutta.io**


March 2 & 3, 2020
From developer to software architect
Presented by **Nathaniel Schutta**

springonetour.io

SpringOne TOUR by Pivotal

## Cloud-Native Java From the Source

The SpringOne Tour brings the best Cloud-Native Java content from our flagship conference directly to you. In 2 days, you'll learn about both traditional monolithic and modern, Cloud-Native Java from the source. Experience valuable facetime with expert Pivotal speakers in both traditional presentation and informal Pivotal Conversations about modern Application Development, DevOps, CI/CD, Cloud and more.