

# CLOUD NATIVE 101

NATHANIEL SCHUTTA  
@NTSCHUTTA  
NTSCHUTTA.IO

Ah “the cloud!”

So. Many. Options.

Microservices. Modular monoliths.

Container all the things?

What about serverless?

# Functions. As a Service.

Is that cloud native?

How do we make  
sense of all this?!?



WHAT IS CLOUD  
NATIVE?

Applications designed to take  
advantage of cloud computing.

Fundamentally about how we  
create and deploy applications.

Cloud computing gives us  
some very interesting abilities.

Scale up. Scale down. On demand.

Limitless compute.\*

\* Additional fees may apply.

Cloud native isn't just an  
architectural pattern.

Combination of practices,  
techniques, technologies.

Agile development.

Continuous delivery.

# Automation.

# Containers.

# Microservices.

# Functions.

Changes our culture.

# DevOps.

Infrastructure is a different  
game today isn't it?

We've seen this massive shift.

Servers used to be home grown.

Bespoke. Artisanal.

Spent days hand crafting them.

Treated them like pets...



Did whatever it took to keep  
them healthy and happy.

Servers were a heavily constrained resource.

They were really expensive!

Had to get our money's worth...

Thus was born app servers.

Put as many apps as  
possible on a server.

Maximize the return on investment.

But that has some  
unintended side effects.

Shared resources.

One application's bug could  
take down multiple apps.

Coordinating changes hurts.

“Your app can’t get this feature  
until all other apps are ready.”

Currency === 18 months of  
freezes, testing, frustration.

Organizations ignored currency  
issues...pain wasn't "worth it".

“Fear is the path to the dark side.  
Fear leads to anger. Anger leads  
to hate. Hate leads to suffering.”

-Yoda

#YodaOps

Move ~~code~~ from one  
server to another...

Worked in dev...but not test.

Why?!?

The environments are  
the same...right?

“Patches were applied in a  
different order.”

Can I change careers?

Things started to change.

Servers became commodities.

Linux and Intel chips replaced  
custom OS on specialized silicon.

Prices dropped.

Servers were no longer the constraining factor.

People costs eclipsed  
hardware costs.

Heroku, AWS, Google App  
Engine, Cloud Foundry, Azure.

Shared servers became a liability.

Treat them like cattle...when  
they get sick, get a new one.

A close-up photograph of the head of a dark-colored horse, likely black or dark brown. The horse has dark eyes and a dark nose. A yellow triangular ear tag is attached to its left ear, featuring the number "15" in green ink. The background is a bright, out-of-focus green field.

15

New abstractions.

Containers and PaaS  
changed the game.

Package the app up with  
everything it needs.

Move **\*that\*** to a  
different environment.

Works in dev? You're testing the  
exact same thing in test.

So. Much. Win.

Your app needs a spiffy  
new library? Go ahead!

It doesn't impact any other app  
because you are isolated.

Moves the value line.

Less “undifferentiated heavy lifting”.

**Onsi Fakhouri**  
@onsijoe

cf push haiku

here is my source code  
run it on the cloud for me  
i do not care how

4:18 PM · May 12, 2015

68 Retweets 105 Likes

<https://mobile.twitter.com/onsijoe/status/598235841635360768?lang=en>

Changes development.

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](#)



The image shows a man in a dark suit standing on a stage, speaking into a microphone. Behind him is a large presentation screen. The screen displays the text "After 10 Months, with Our PAAS PCF Platform" at the top. Below this, there are three red statistics: "29 teams", "4 countries", and "3000+ deploys/month". The Scotiabank logo is visible at the bottom right of the screen. The stage has a white and red color scheme.

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

Speed matters.

Disruption impacts **every** business.

Your industry is not immune.

Amazon Prime customers can  
order from Whole Foods.

Some insurance companies  
view Google as a competitor.

We're all technology  
companies today.

The background of the image is a clear blue sky. It is dotted with various types of clouds, including large, fluffy cumulus clouds and smaller, wispy cirrus clouds. The lighting suggests a bright day, with the clouds appearing white or light grey against the blue sky.

# 12 FACTORS

# Twelve Factor App.

<https://12factor.net>

Characteristics shared by  
successful apps.

At least at Heroku.

I. One codebase in version control, multiple deploys.

Version control isn't  
controversial. Right?!?

Sharing code? It better  
be in a library then...

II. Explicitly define your dependencies.

Do not rely on something just  
“being there” on the server.

If you need it, declare it.

III. Configuration must be  
separate from the code.

The things that vary from  
environment to environment.

Could you open source  
that app right now?

IV. Backing services are just attached resources.

Should be trivial to swap out a local database for a test db.

In other words, loose coupling.

V. Build, release, run.

Deployment pipeline anyone?

Build the executable...

Deploy the executable with the  
proper configuration...

Launch the executable in a  
given environment.

# VI. Stateless - share nothing.

State must be stored via some kind of backing service.

In other words, you cannot rely  
on the filesystem or memory.

Recovery. Scaling.

## VII. Export services via port binding.

App exports a port, listens for incoming requests.

**Localkost** for development,  
load balancer for public facing.

# VIII. Scale via process.

In other words, scale horizontally.

IX. Start up fast, shut  
down gracefully.

Processes aren't pets,  
they are disposable.

Processes can be started (or stopped) quickly and easily.

Ideally, start up is seconds.

Also can handle  
unexpected terminations!

X. Dev/Prod parity.

From commit to production  
should be hours...maybe days.

Definitely not weeks.

Developers should be involved  
in deploys and prod ops.

Regions should be identical. Or  
as close as possible to identical.

Backing services should be the  
same in dev and prod.

Using one DB in dev and  
another in prod invites pain.

# XI. Logs as event streams.

Don't write logs to the filesystem!

It won't be there later...

Write to `stdout`.

Stream can be routed any  
number of places.

And then consumed via a  
wide variety of tools.

XII. Admin tasks run as  
one off processes.

Database migrations for instance.

REPL for the win.

Run in an identical environment  
to the long running processes.

Your legacy apps will  
violate some factors.

Maybe all 12!

In general...

II. Explicitly define your dependencies.

Probably one of the  
harder ones to satisfy.

Do we **really** need this library?

“It works, don’t touch it.”

III. Configuration must be  
separate from the code.

Many an app has  
hardcoded credentials.

Hardcoded database connections.

# VI. Stateless - share nothing.

Also can be challenging.

Many apps were designed  
around a specific flow.

Page 2 left debris for Page 3!

“Just stash that in session”.

IX. Start up fast, shut  
down gracefully.

Many apps take way  
too long to start up...

Impacts health checks.

X. Dev/Prod parity.

Environments should be consistent!

Shorten code to prod cycle.

“It worked in test...”

Do your applications have to be  
fully 12 factor compliant?

Nope.

Is it a good goal?

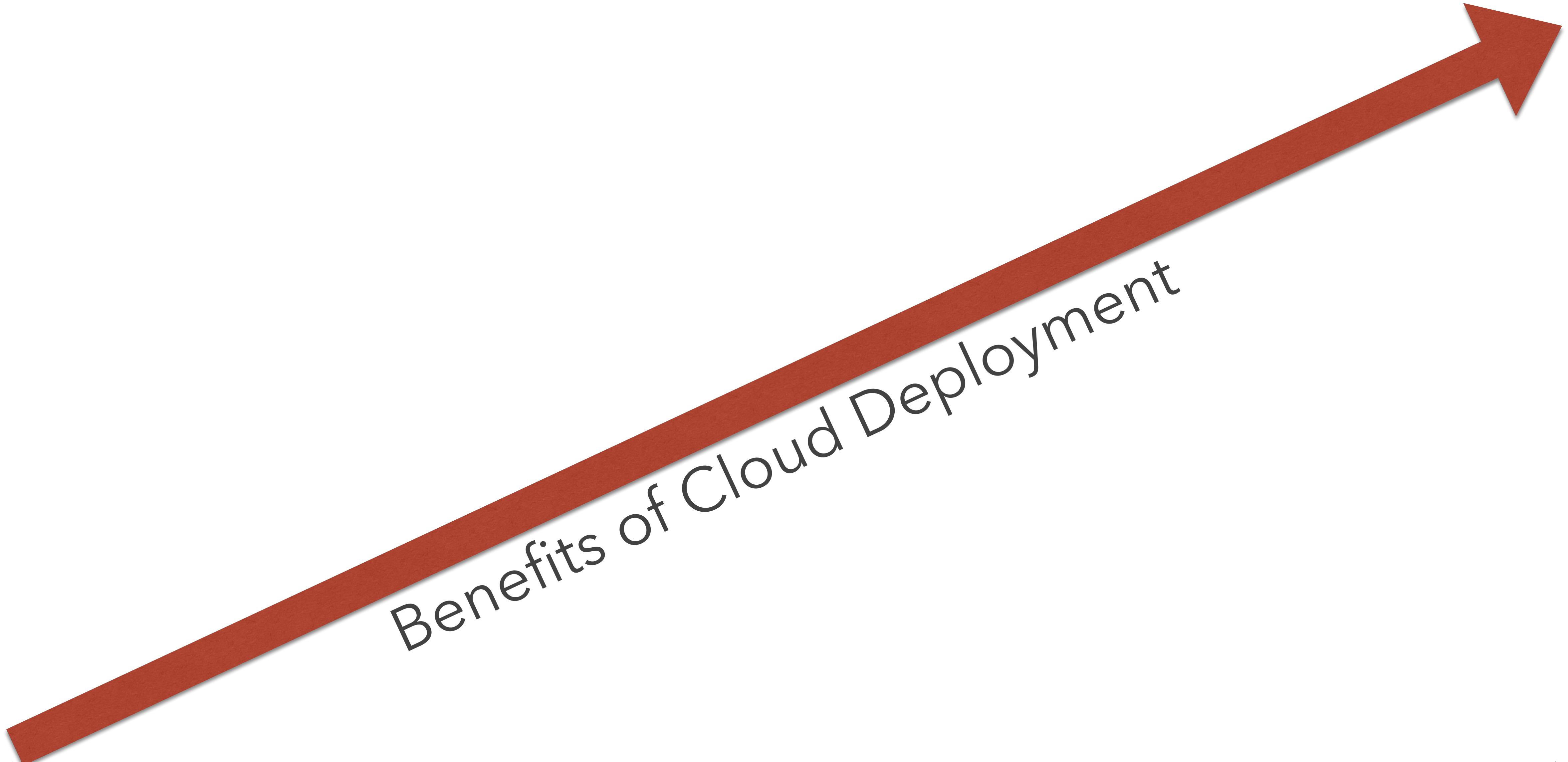
Sure.

But be pragmatic.

Certain attributes lessen the  
advantages of cloud.

Long startup time hurts elastic  
scaling & self healing.

Think of it as a continuum.



Benefits of Cloud Deployment



12 Factor Compliance

Developers also talk  
about 15 factor apps.

aka Beyond the Twelve-Factor App.

<https://content.pivotal.io/blog/beyond-the-twelve-factor-app>

However you define it...

To maximize what  
the cloud gives us...

Applications need to be  
designed properly.

Legacy applications will fall short.

Opportunistically refactor!

Building greenfield?

Go cloud native!

Don't build legacy.



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...



<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.

# Please Microservice Responsibly.

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

“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](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>

The background of the image is a wide-angle photograph of a natural landscape. At the bottom, there's a dark blue body of water, possibly a lake or sea. In the middle ground, a small, dark island or peninsula is visible in the distance. The upper two-thirds of the image are dominated by a vast sky filled with various types of clouds. Large, billowing cumulus clouds are scattered across the scene, with some appearing bright white and others darker grey. The sky transitions from a deep blue at the top to a lighter, more overcast tone near the horizon.

SERVERLESS

From IaaS to CaaS to PaaS...

What about serverless?

# Functions.

As a Service.

I hear that is \*the\* in thing now.

But we just refactored to cloud  
native microservices...



( )

—

Don't throw that code away just yet!

Fair to say FaaS is a  
subset of serverless.

Though many use the  
terms interchangeably.

First things first. There  
are still servers.

We are just (further)  
abstracted away from them.

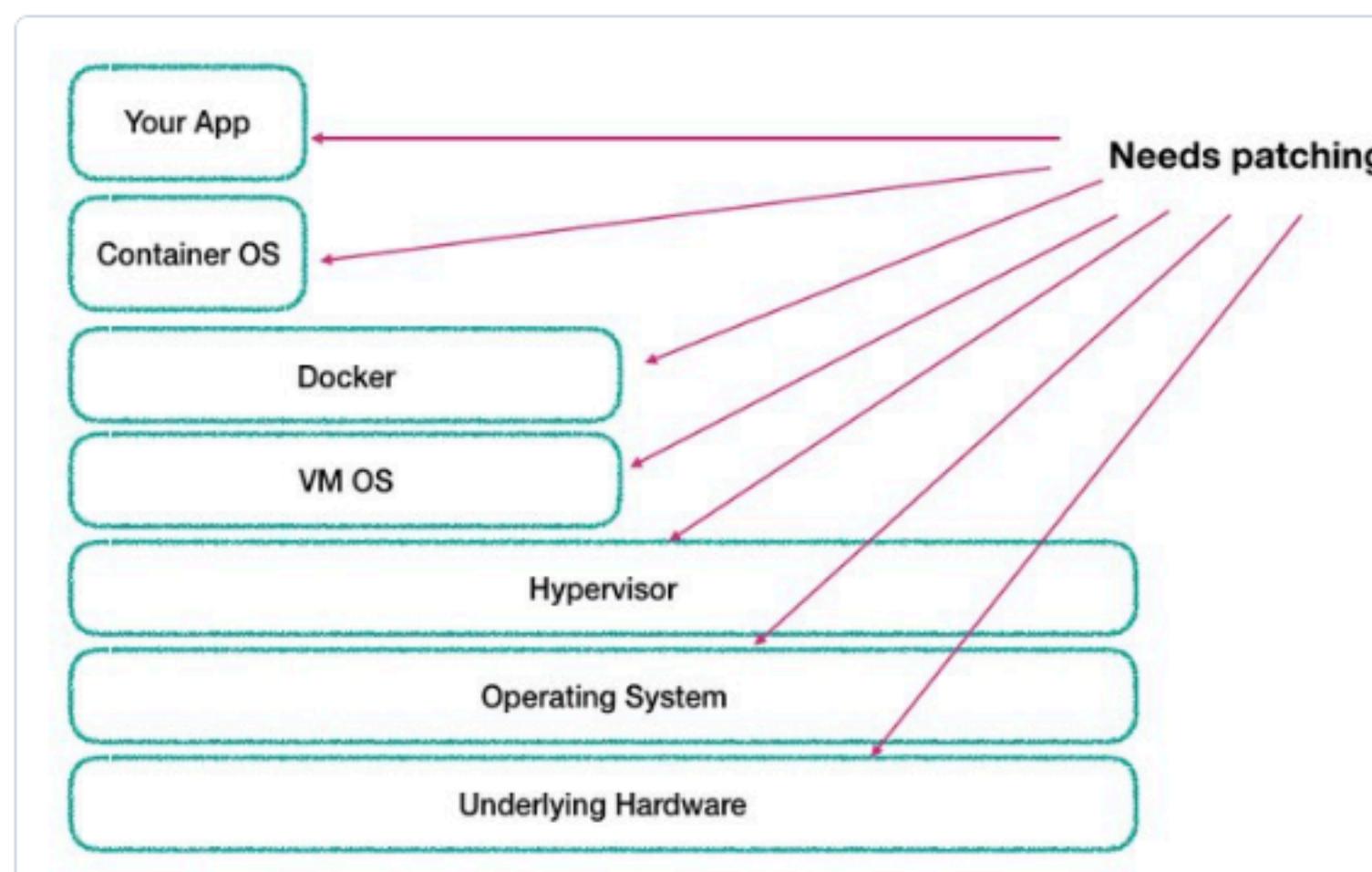
We don't have to spend time  
provisioning, updating, scaling...

In other words it is  
someone else's problem.



Sam Newman ✅  
@samnewman

I was in the middle of creating this slide (wrt patch hygiene) and had to stop half-way through and ask myself - aren't we all just making this worse?



12:35 PM · Jan 14, 2018

1,071 Retweets 1,640 Likes

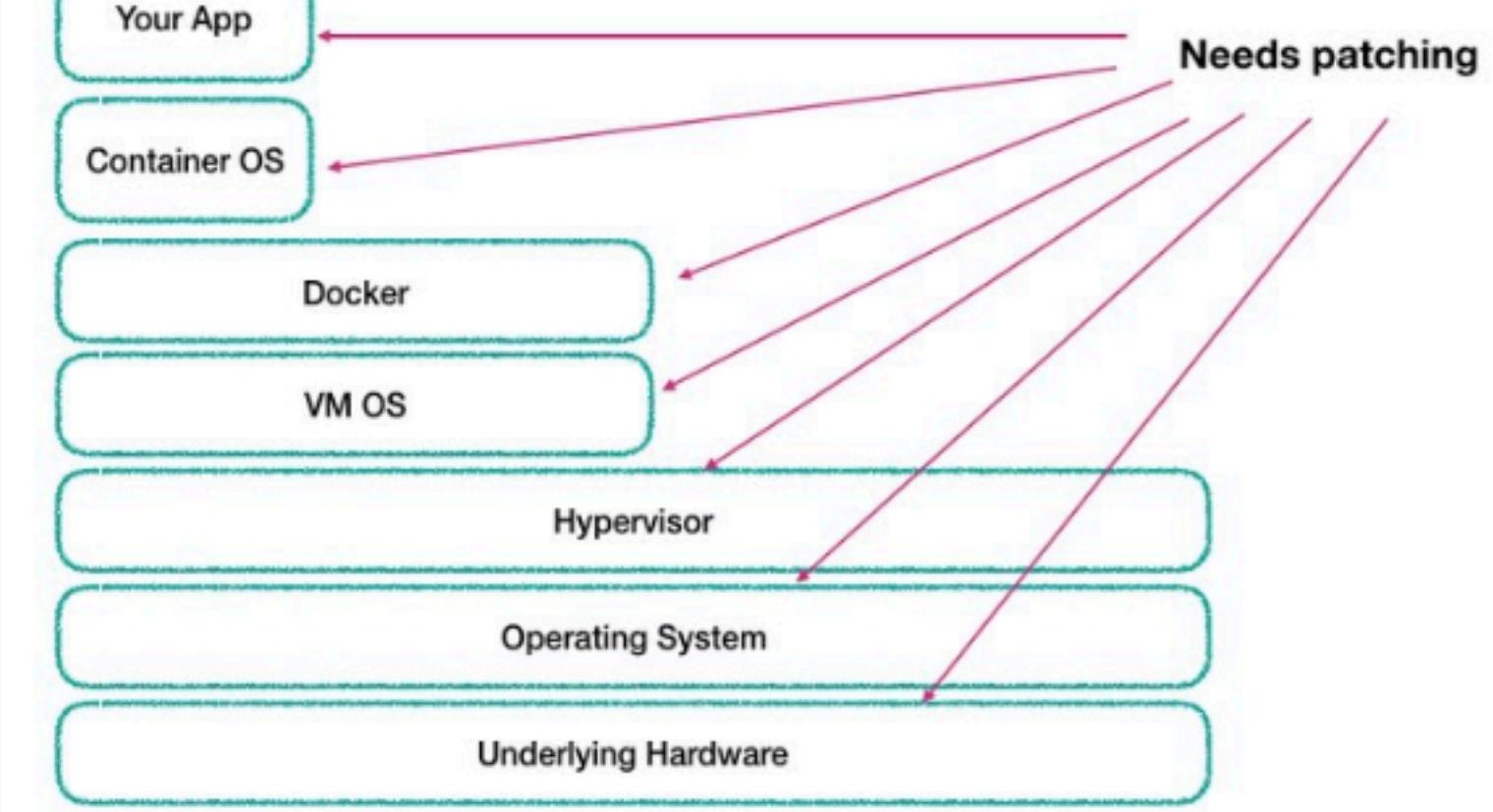


<https://mobile.twitter.com/samnewman/status/952610105169793025>

 **Josh Long (龙之春, जोश) ✅**  
@starbuxman

This is exactly why an integrated platform like [@cloudfoundry](#) & a public cloud offering is valuable: everything in that slide (except your app) is either managed centrally by the platform or its managed by somebody else who have a vested interest in doing so well.

**Sam Newman ✅** @samnewman  
I was in the middle of creating this slide (wrt patch hygiene) and had to stop half-way through and ask myself - aren't we all just making this worse?  
[Show this thread](#)



```
graph TD; YourApp[Your App] -- "Needs patching" --> ContainerOS[Container OS]; ContainerOS -- "Needs patching" --> Docker[Docker]; Docker -- "Needs patching" --> VMOS[VM OS]; VMOS -- "Needs patching" --> Hypervisor[Hypervisor]; Hypervisor -- "Needs patching" --> OS[Operating System]; OS -- "Needs patching" --> HW[Underlying Hardware]
```

4:03 AM · Feb 2, 2018

17 Retweets 35 Likes

Comment Share Heart Envelope

<https://mobile.twitter.com/starbuxman/status/959366771462496256>

# Containers

Developer  
Provided

Container

Container  
Scheduling

Primitives for  
Networking,  
Routing, Logs and  
Metrics

# Platform

Application

Container

Container Images

L7 Network

Logs, Metrics,  
Monitoring

Services  
Marketplace

Team, Quotas &  
Usage

# Serverless

Function

Container

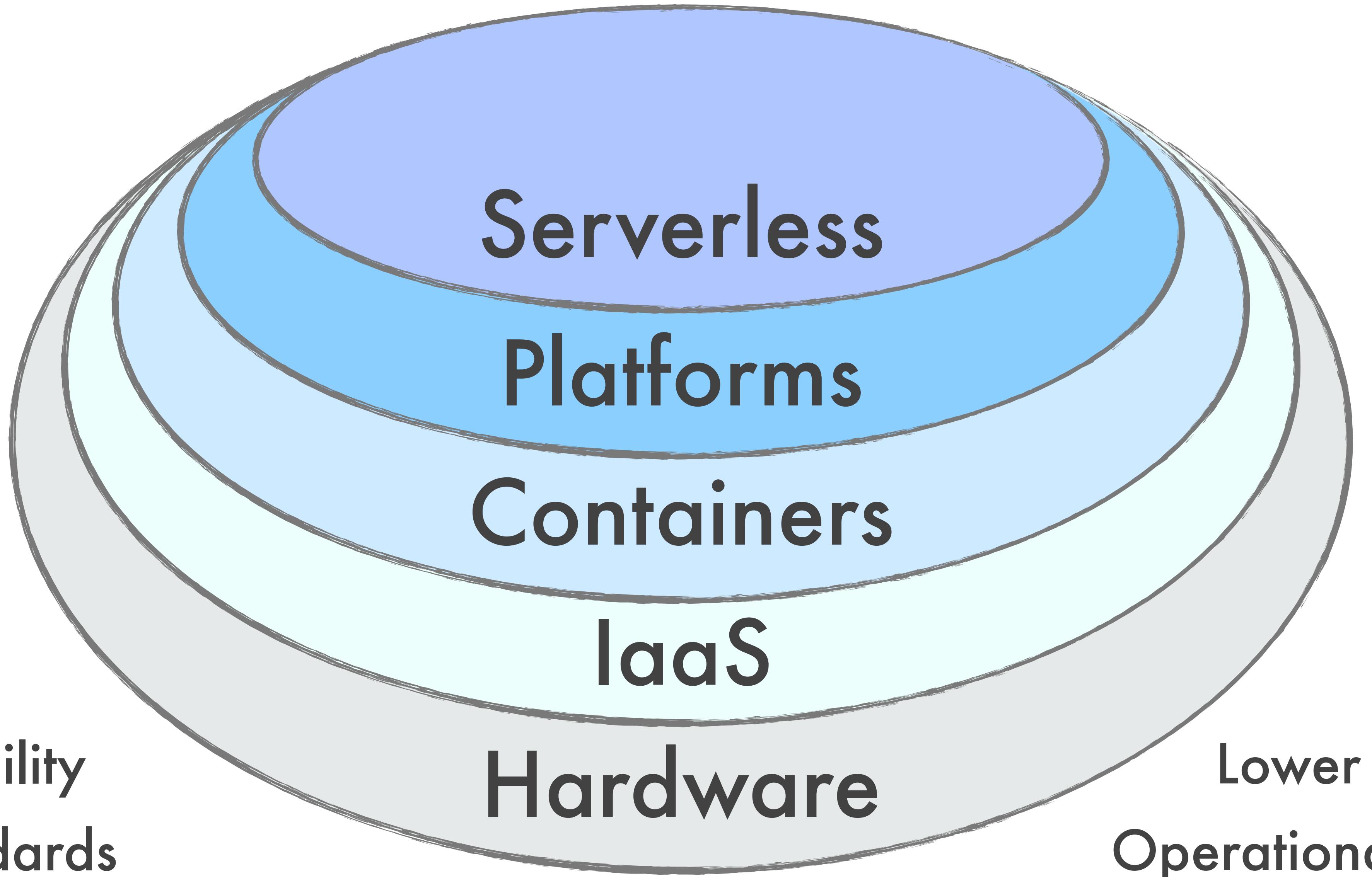
Function Execution

Function Scaling

Event Stream  
Bindings

IaaS

Different levels of abstraction.



Push as many workloads up the stack as feasible.

Veritable plethora of options.

AWS Lambda, Azure Functions,  
Google Cloud Functions...

riff, OpenWhisk, Kubeless, Fission...

Definitely suffers from the  
shiny new thing curse.



And everything that entails.



There **\*are\*** very good reasons  
to utilize this approach!

Some relate specifically  
to a platform.

Perhaps you already have  
data with a provider.

Using Amazon S3 for example.

You might want to tap into particular services offered by a provider.

Perhaps you want to utilize Google  
Cloud Machine Learning Engine.

But it isn't just a new a way to cloud.

There are serious efficiency gains  
to be had with this approach!

Development efficiencies.

Functions push us further up  
the abstraction curve.

Allows us to focus on  
implementation not infrastructure.

Do you know what OS your  
application is running on?

Do you care?

What \*should\* you care about?

Where is the “value line” for you?

We want to get out of the business  
of “undifferentiated heavy lifting”.

Focus on business problems,  
not plumbing problems.

Resource efficiencies.

Function hasn't been  
called recently?

Terminate the container.

Request comes in? Instance  
springs into existence.

Every 3.0s: kubectl get functions,topics,pods,services,deploy Firethorn.local: Mon Jan 8 15:23:28 2018

NAME	AGE
functions/echo-java	32s

NAME	AGE
topics/echo-java	32s

NAME	READY	STATUS	RESTARTS	AGE
po/demo-riff-function-controller-6975dbdc7d-kvfxm	1/1	Running	5	5h
po/demo-riff-http-gateway-64fc56bd96-vmvnn	1/1	Running	11	5h
po/demo-riff-kafka-c7f456685-p65bv	1/1	Running	9	5h
po/demo-riff-topic-controller-58694bb5bf-n5fb5	1/1	Running	5	5h
po/demo-riff-zipkin-6c66b788bb-dhsjm	1/1	Running	0	1h
po/demo-riff-zookeeper-6fd5c5bd54-fqvvk	1/1	Running	2	5h

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/demo-riff-function-controller	ClusterIP	10.109.143.74	<none>	80/TCP	5h
svc/demo-riff-http-gateway	NodePort	10.101.136.125	<none>	80:31718/TCP	5h
svc/demo-riff-kafka	ClusterIP	10.96.180.243	<none>	9092/TCP	5h
svc/demo-riff-zipkin	LoadBalancer	10.98.172.131	<pending>	9411:31239/TCP	1h
svc/demo-riff-zookeeper	ClusterIP	10.110.26.126	<none>	2181/TCP	5h
svc/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	38d

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/demo-riff-function-controller	1	1	1	1	5h

Functions aren't free however.

Well, ok, once you get beyond  
the free tier at least...

Then some fractional  
cost there after.

First million (or two)  
requests are free\*.

\* Additional fees may apply.

For example: data transfer fees  
or other services you leverage.

Charged based on # of requests,  
run duration & resource allocation.

Can be challenging to determine  
just how much it will cost...

But for certain workloads,  
it is very cost effective.

Operational efficiencies.

# Serverless ops?

Again, less for us to worry about.

Rely on a platform.

What use cases fit well  
with functions?

Rapidly evolving  
business requirements...

Stateless workloads.

Infrequent (or sporadic) requests.

Variable scaling needs.

Asynchronous workloads.

Easy to parallelize.

IoT, machine learning, log  
ingestion, batch processing.

“Conversational UIs” aka  
digital assistants like Alexa.

CI/CD automation, chat  
integration, stream processing...

Website back end services such as  
logging, post handlers, auth.

Event processing, monitoring,  
notifications, alerting.

Security scanning.

Very valuable tool.

It isn't a good fit for every workload.

But you knew that.

# DEVOPS



This can all seem a bit...  
overwhelming.



**CHANGE BAD!**

A day in the life...

Tools will change.

Culture will change.

Must evolve past “DevOps fills  
out tickets for developers”.

Site Reliability Engineers.

The traditional sys admin approach  
doesn't give us reliable services.

Inherent tension.

Conflicting incentives.

Developers want to release  
early, release often.

Always Be Changing.

But sys admins want stability.

It works. No one touch anything.

Thus trench warfare.

Doesn't have to be this way!

We can all get along.

What if we took a different  
approach to operations?

“what happens when you  
ask a software engineer to  
design an operations team.”

<https://landing.google.com/sre/book/chapters/introduction.html>

Ultimately, this is just software  
engineering applied to operations.

Replace manual tasks  
with automation.

Focus on engineering.

Many SREs are software engineers.

Helps to understand UNIX  
internals or the networking stack.

Our operational  
approach has to evolve.

The “Review Board” meeting  
once a quarter won’t cut it.

How do we move fast safely?

Operations must be able to  
support a dynamic environment.

That is the core of what we mean  
by site reliability engineering.

How we create a stable, reliable  
environment for our services.

It doesn't happen in spare cycles.

Make sure your SREs have time  
to do actual engineering work.

On call, tickets, manual tasks -  
shouldn't eat up 100% of their day.

SREs need to focus on automating  
away “toil” aka manual work.

Isn't this just DevOps?

Can argue it is a natural extension of the concept.

Think of SRE as a specific  
implementation of DevOps.

Technology changes.  
Feature, not a bug.

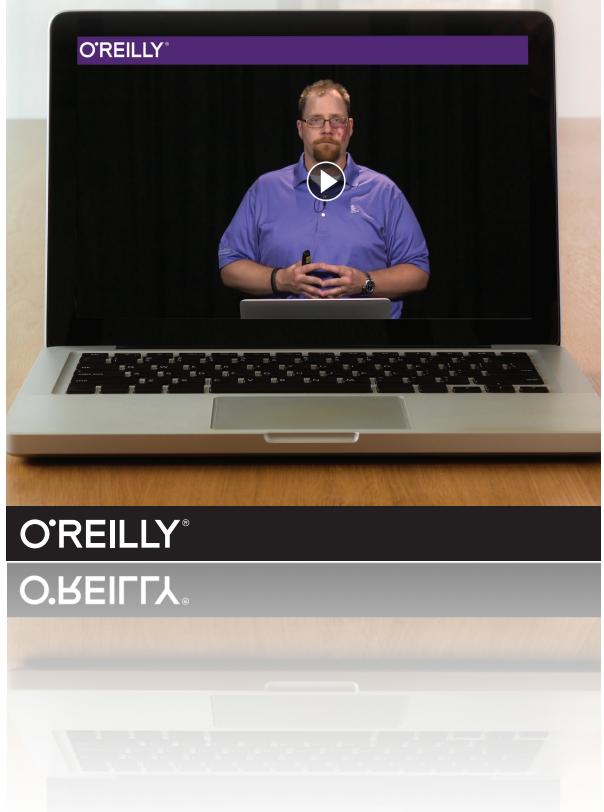
We need to evolve along with it.

The more things change, the  
more they stay the same.

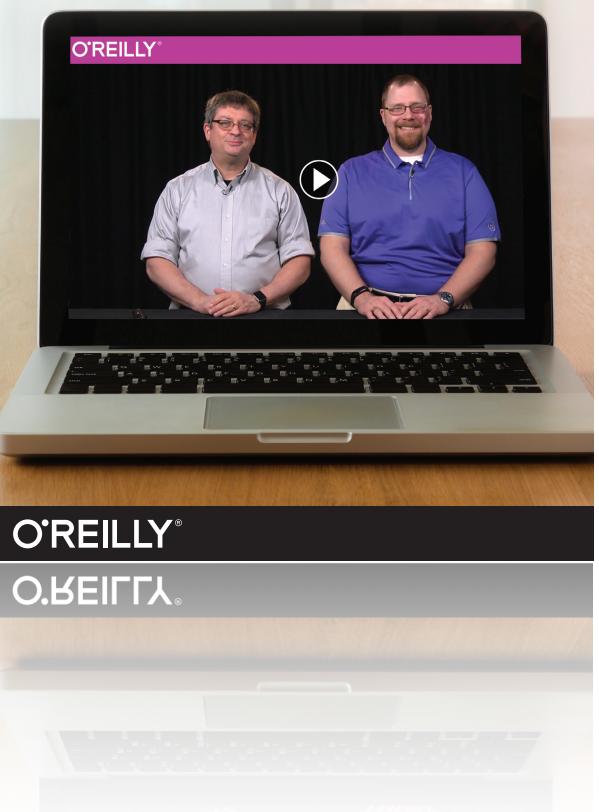
Good luck!

# Thanks!

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



Presentation  
Patterns  
*with Neal Ford & Nate Shutta*



Modeling for  
Software  
Architects  
*with Nate Shutta*



Nathaniel T. Schutta  
[@ntschutta](https://twitter.com/ntschutta)  
[ntschutta.io](http://ntschutta.io)

