*aaS

CLOUD NATIVE
COMPUTING FOUNDATION

kubernetes

#abstractions

#microservices

Presentation by

MEDIA
NETWORKS
LABORATORY

HISTORY

Containers

PRESENTATION STRUCTURE

Kubernetes

Conclusions

HISTORY

IAAS *PAAS* *SAAS* *BAAS* *MAAS* *FAAS* *CLOUD-NATIVE* *SERVERLESS*

https://loige.co/from-bare-metal-to-serverless/
https://hackernoon.com/why-the-fuss-about-serverless-4370b1596da0


THE
BARE METAL
AGE

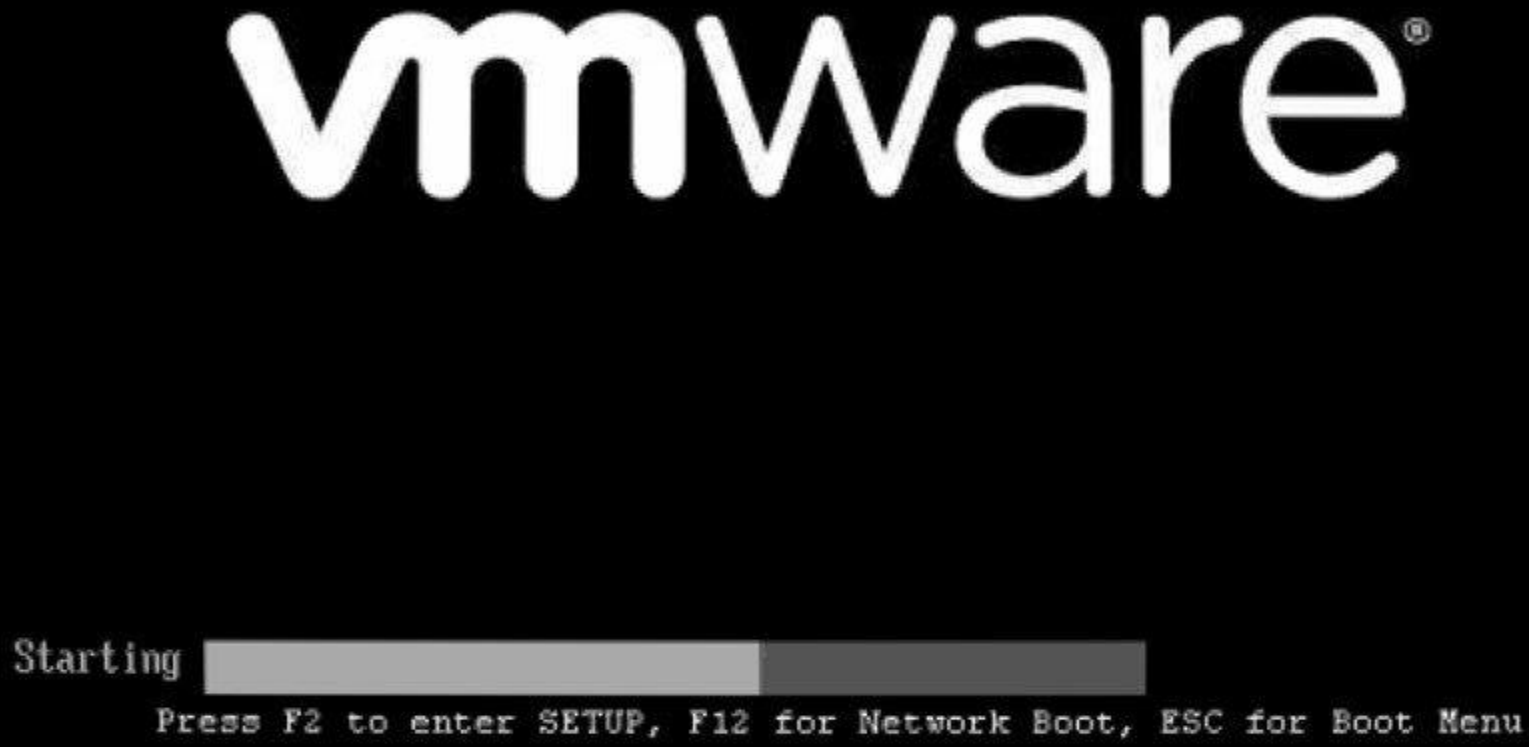
Marc Benioff, 1999
launching Salesforce

MTTR

mean time to recovery

N+1 multiple power-sources capacity planning

2001



2001, VMWare releases ESXi
and server virtualization becomes a thing


THE
IaaS
AGE
2006
2002-2006 Amazon Web Services
EC2 (Virtual Machine service), S3 (Scalable storage service) and SQS (message queuing system)

THE

PaaS

AGE

2009

2009, Adam Wiggins
Heroku

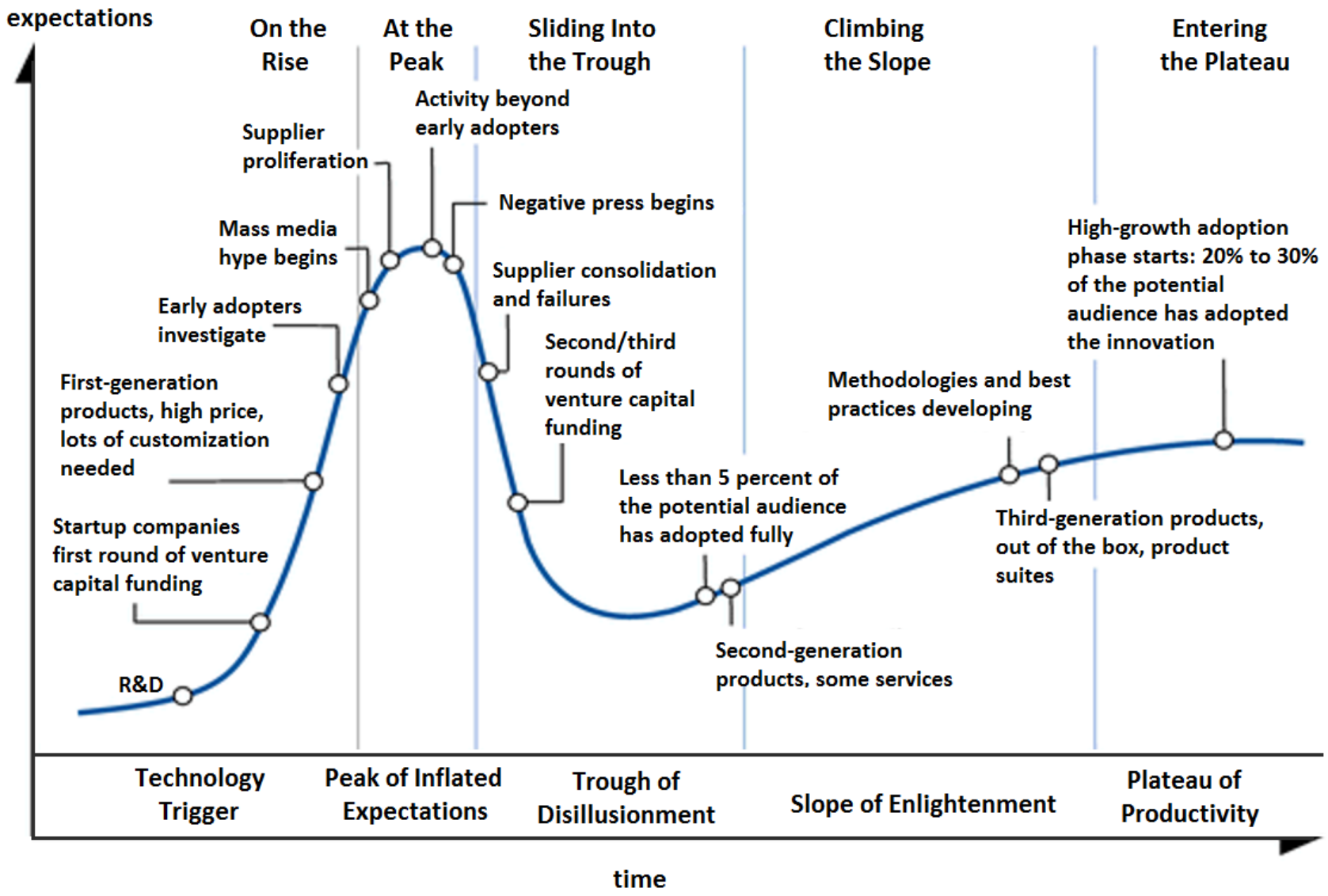
Database-as-a-Service

2011

2011, James Tamplin
Firebase

Hype-cycle

2012



Pets or Cattle?

2012

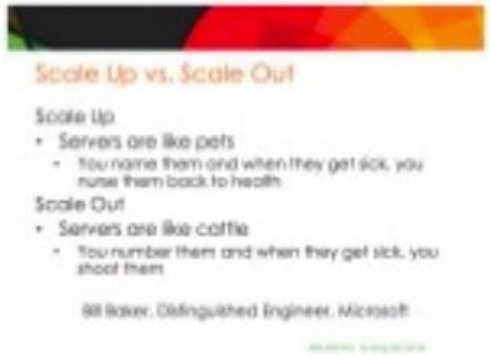
How to Explain Why Cloud is Different

Hint: **it's about uniqueness**

Scale-up or Scale-out?

2012

Microsoft



Cloudscaling

CERN

IBM

Scalr

Rackspace

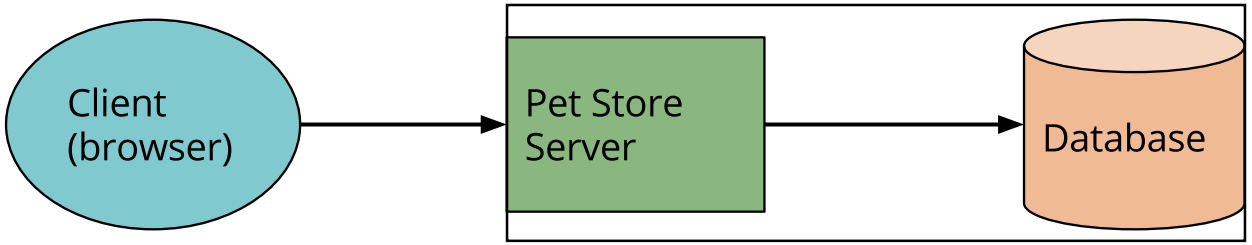
Red Hat

2012

> In the old way of doing things, we treat our servers like pets. For example, 'Frodo' the mail server. If Frodo goes down, it's all hands on deck. The CEO can't get his email and it's the end of the world.
>
> In the new way, servers are numbered, like cattle in a herd. For example, www001 to www100. When one server goes down, it's taken out back, shot, and replaced on the line.

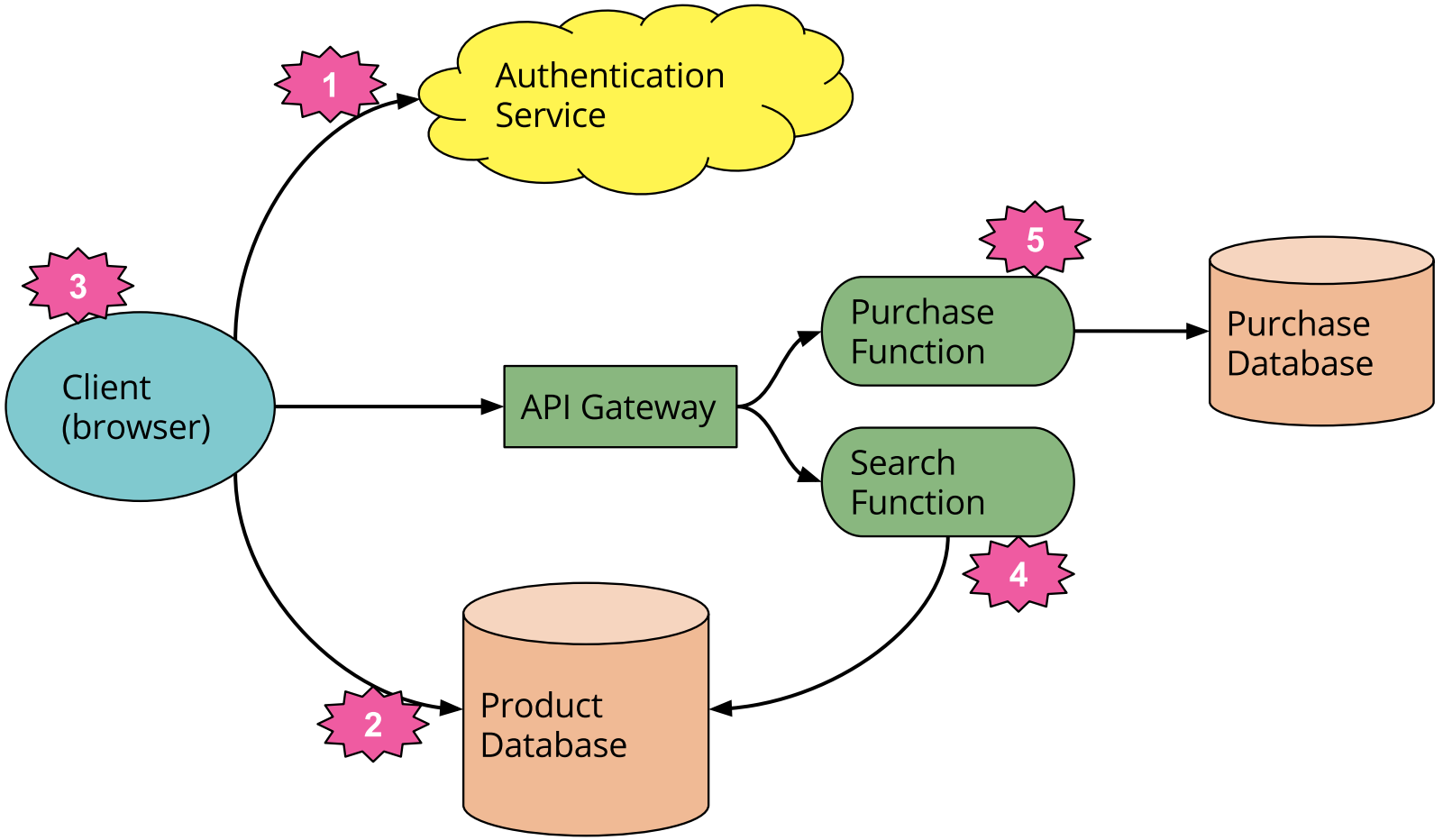
— Randy Bias (many hundreds of times from 2012-2016)

We still haven't fixed
the **deployment times**!



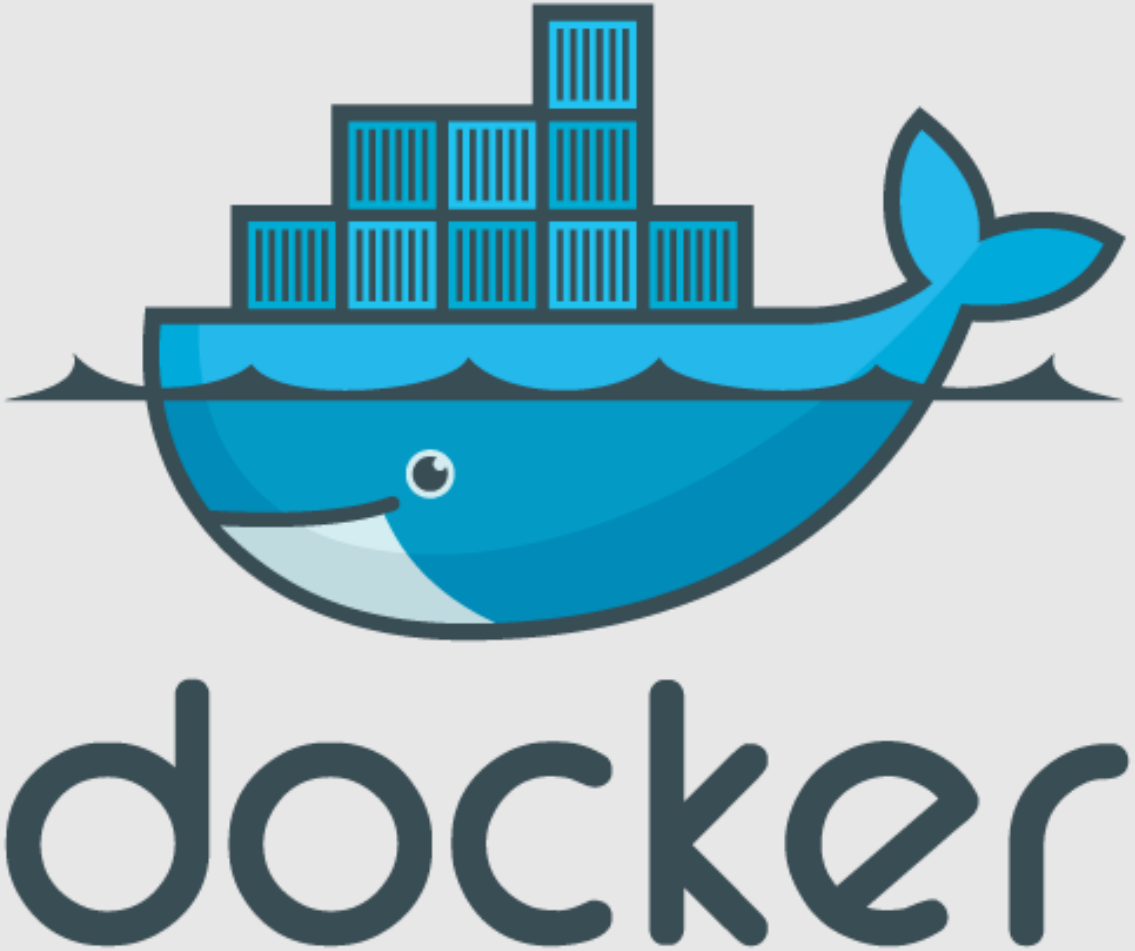
apps are monoliths
and this is a problem

Separation of concerns / Microservices



Containers…

2013



2013, Solomon Hykes
Docker

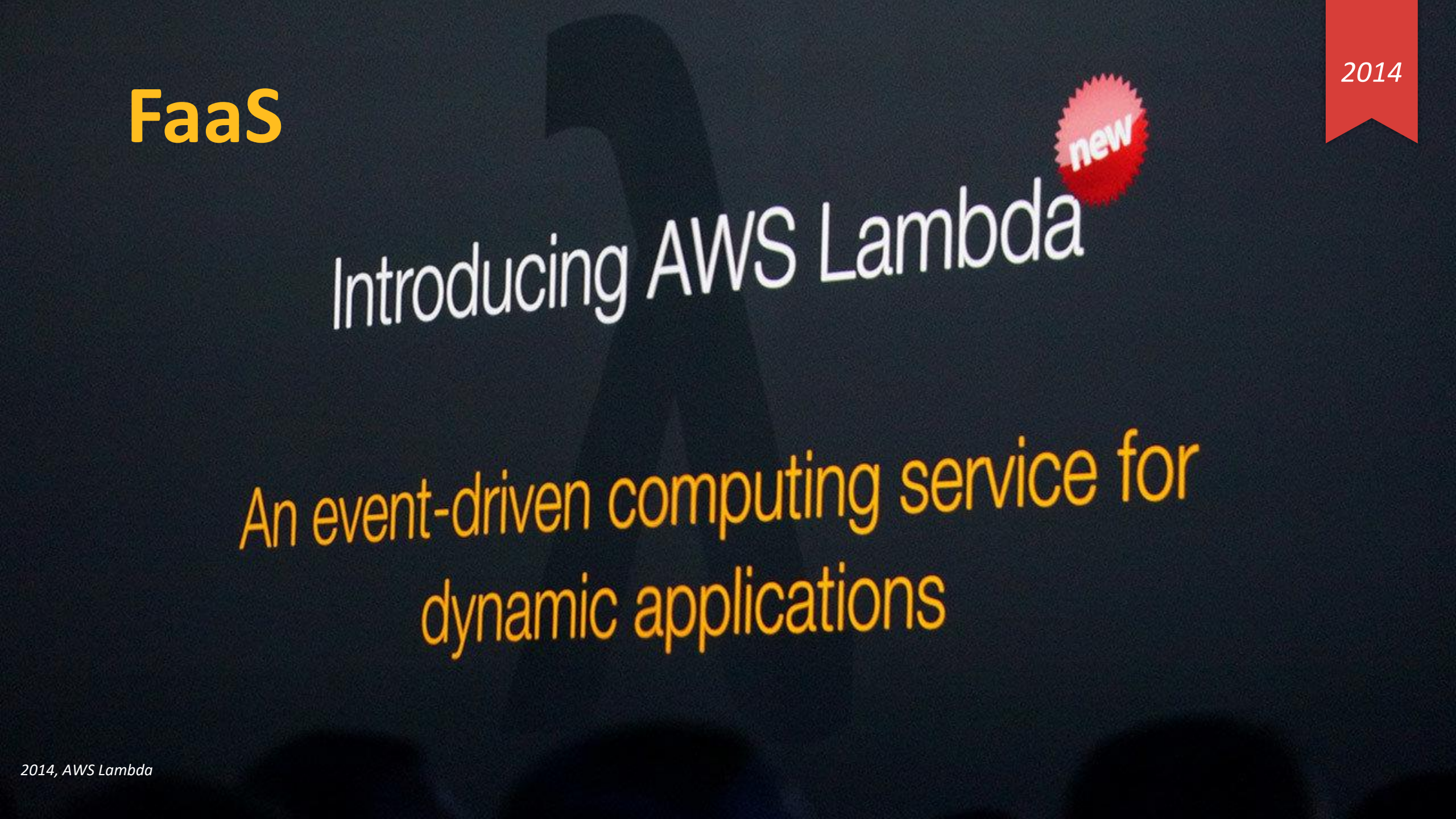
Containers at scale: Kubernetes

2014

2014, Kubernetes

FaaS

2014



2014, AWS Lambda

2018

Attributes of a Serverless Product

- **No management of Server Hosts or Processes**
- **Self auto provision & auto-scale based on load**
- **Costs based on actual, precise, usage**

June
2018

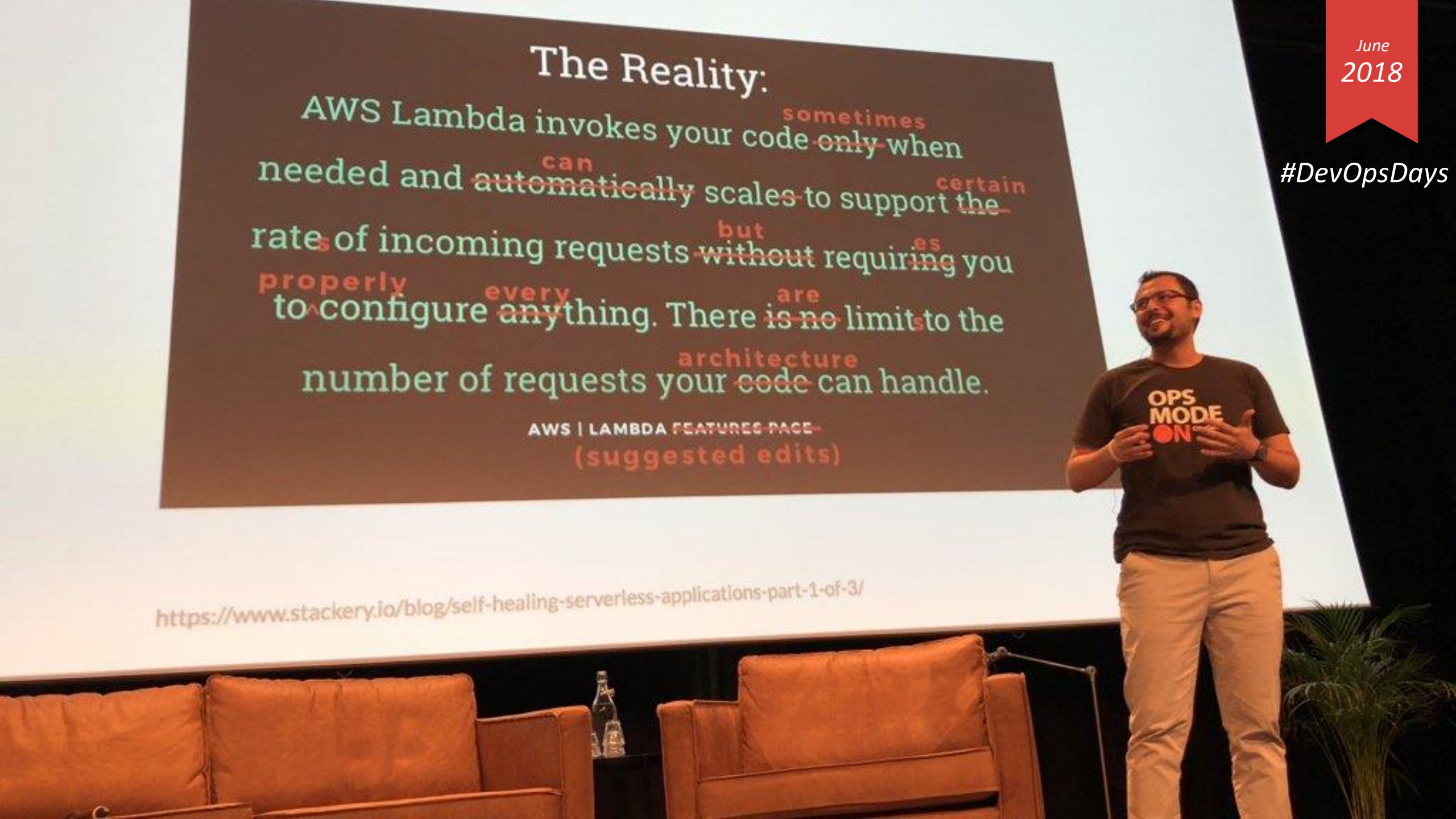
#DevOpsDays

The Reality:

AWS Lambda invokes your code ~~only~~ sometimes when needed and ~~automatically~~ can scales ~~to~~ certain support ~~the~~ rate~~s~~ of incoming requests ~~without~~ but requiring~~s~~ you to configure ~~anything~~ properly every. There ~~is no~~ are limit~~s~~ to the number of requests your ~~code~~ architecture can handle.

AWS | LAMBDA ~~FEATURES PAGE~~ (suggested edits)

https://www.stackery.io/blog/self-healing-serverless-applications-part-1-of-3/



Deployment Abstractions

Less abstracted ⟷ More abstracted

Concept	"Bare metal" servers	Virtual private servers (VPS)	Infrastructure-as-a-service (IaaS)	Platform-as-a-Service (PaaS)	Serverless Compute
Implementation Examples	Dell, Lenovo, Build-your-own	Linode, Digital Ocean, OVH	AWS, Azure, Google Cloud	Heroku, CodeStar, PythonAnywhere	AWS Lambda, Azure Functions

The further right means giving more control and trust to a platform. There are tradeoffs along the entire deployment spectrum so left or right is not inherently better.

Cloud-native?

Serverless?

What is it all about?

*aaS?

It's all about abstractions

@samnewman

It’s a developer-focused wave of services

 follows…

Containers

Treating containers like a black box will eventually leave you in the dark.
@kelseyhightower

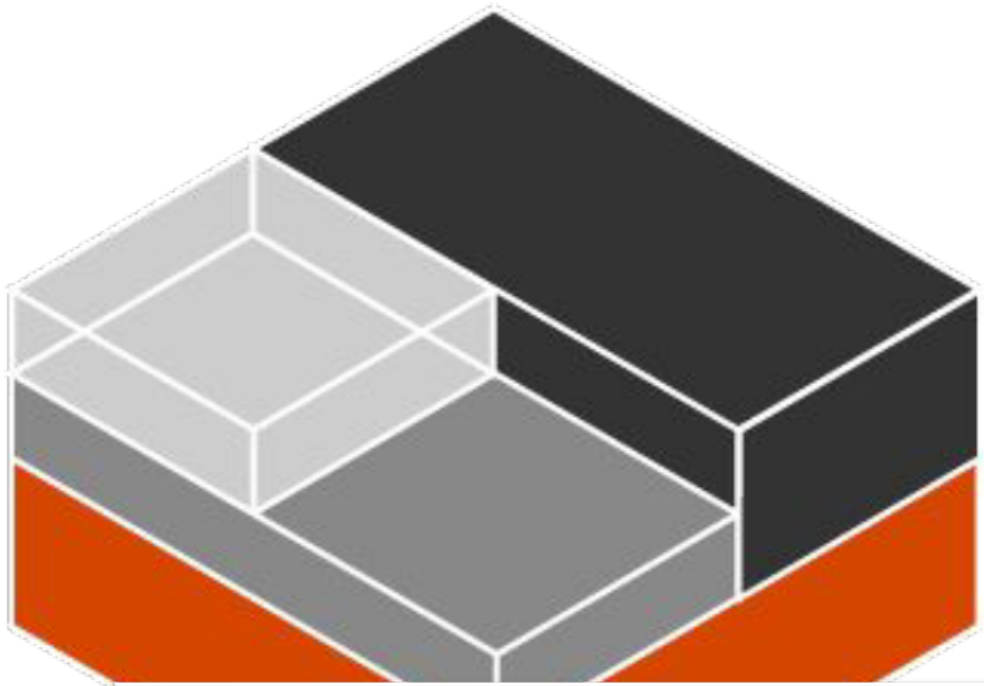
High level approach: it's a lightweight VM

- I can get a shell on it (through SSH or otherwise)
- It "feels" like a VM:
  - own process space
  - own network interface
  - can run stuff as root
  - can install packages
  - can run services
  - can mess up routing, `iptables` ...

Low level approach: it's chroot on steroids

- It's not quite like a VM:
  - uses the host kernel
  - can't boot a different OS
  - can't have its own modules
  - doesn't need `init` as PID 1
  - doesn't need syslogd, cron...
- It's just a bunch of processes visible on the host machine (contrast with VMs which are opaque)

Linux Containers



Kernel namespaces: sandboxing processes from one another

Control Groups (cgroups): control process resource allocations

Security: capabilities drop (seccomp), Mandatory access control (SELinux, Apparmor)

Cgroups = limits how much you can use;
namespaces = limits what you can see (and therefore use)

@jorgemoralespou

LXC

…is a system container runtime designed to execute "full system containers", which generally consist of a full operating system image. An LXC process, in most common use cases, will boot a full Linux distribution such as Debian, Fedora, Arch, etc, and a user will interact with it similarly to how they would with a Virtual Machine image.

LXD

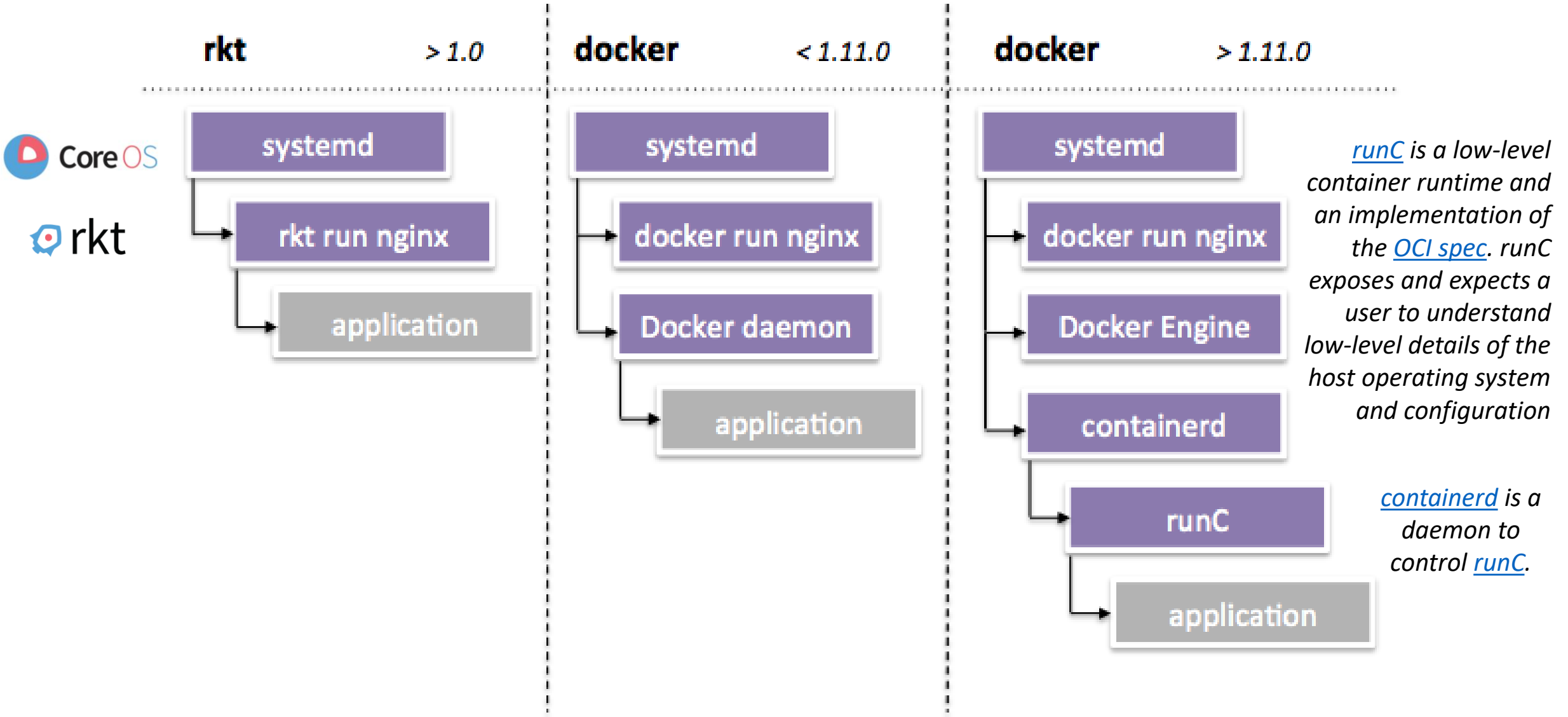
…is similar to LXC but is a REST API on top of liblxc which forks a monitor and container process. This ensures the LXD daemon is not a central point of failure and containers continue running in case of LXD daemon failure. All other details are nearly identical to LXC.

Container Wars

“Docker’s architecture is fundamentally flawed”

At the heart of Docker is a daemon process that is the starting point of everything Docker does. The docker executable is merely a REST client that requests the Docker daemon to do its work. Critics of Docker say **this is not very Linux-like**.

Container Wars

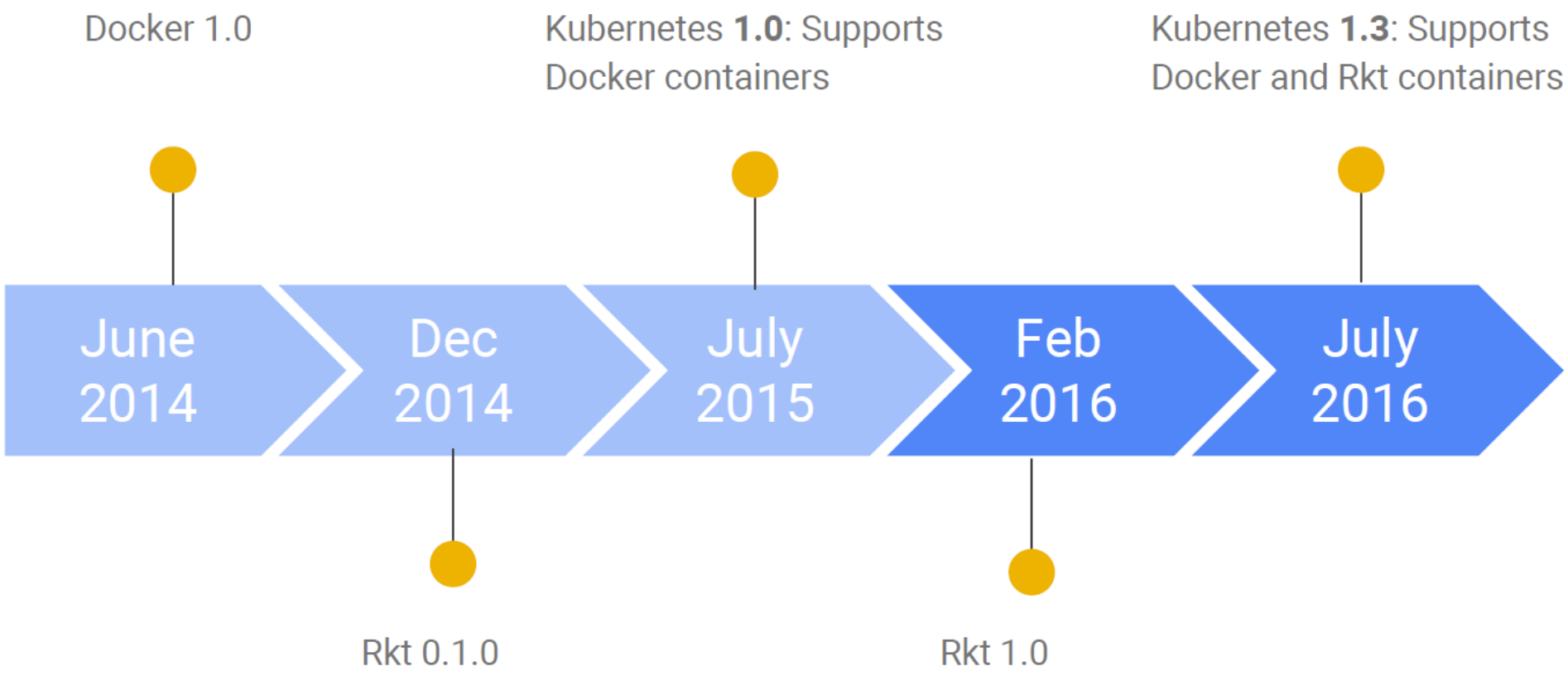


Kubernetes

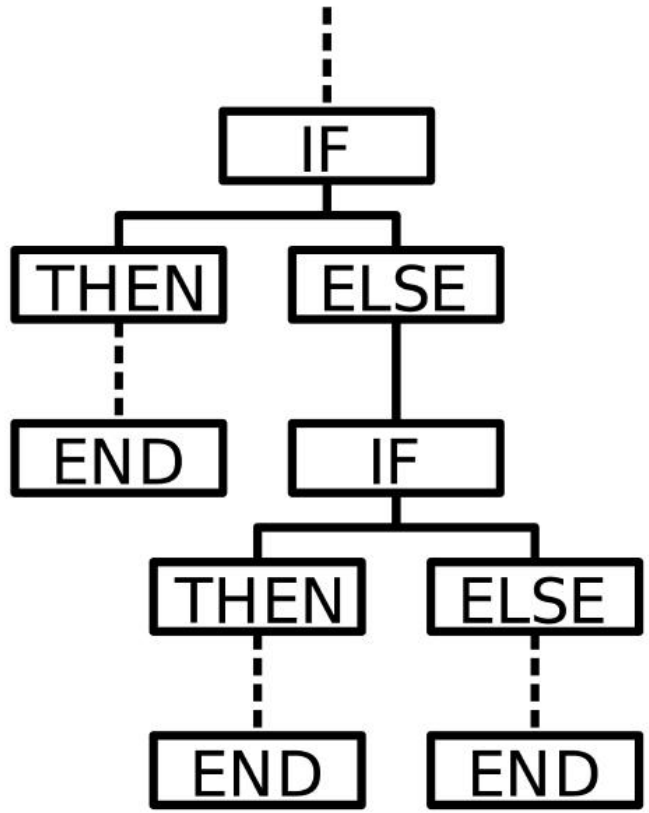
Kubernetes is a container management system

Kubernetes is a container management ~~system~~ platform

Remember Container runtimes?



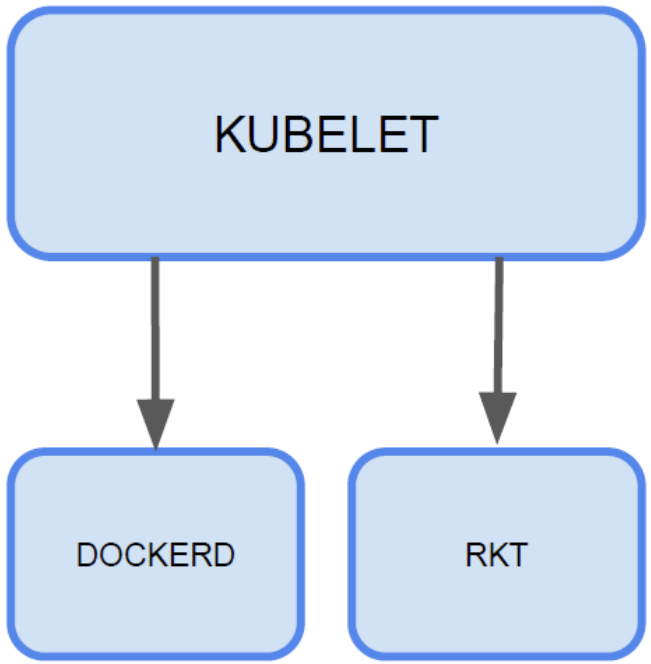
and code got messy

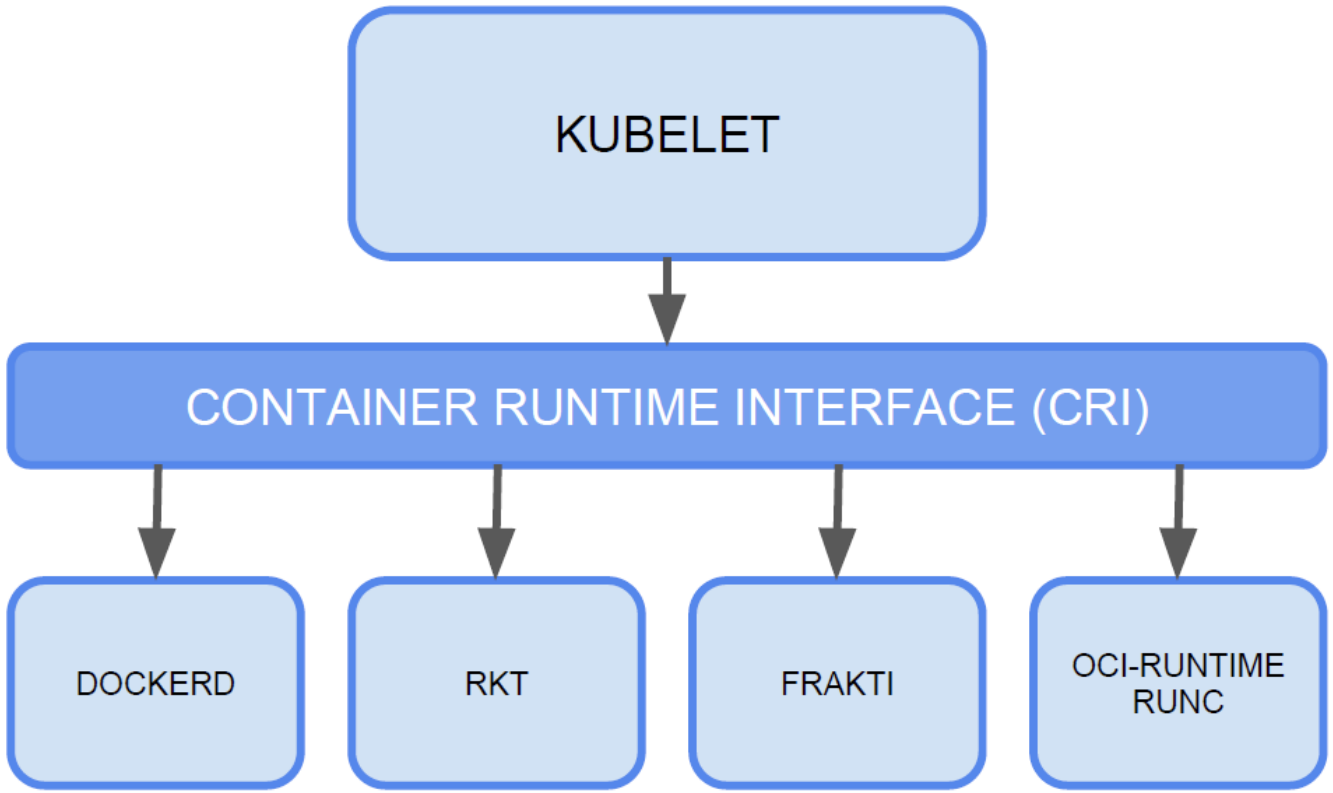


Standardize containers

- **Runtime spec (runc = Reference implementation)**
- **Image spec**
- **Distribution spec (proposal)**

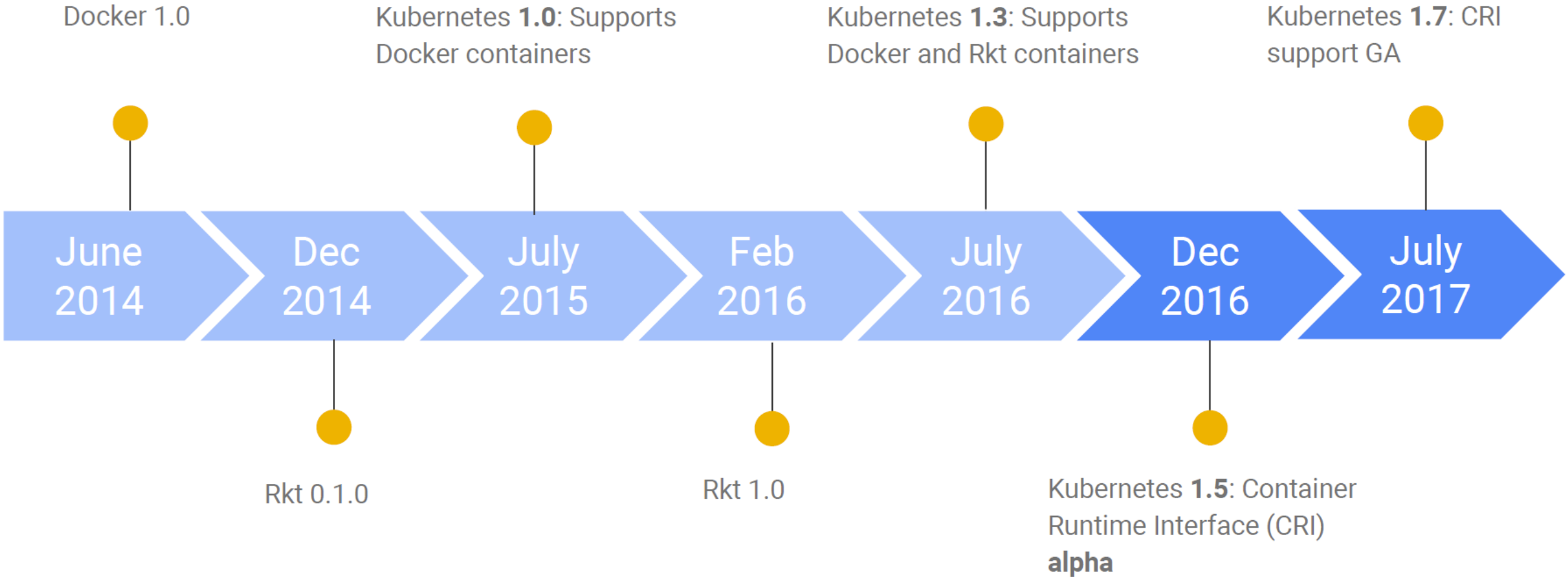
Use API/Interfaces to Container Runtimes





@jorgemoralespou

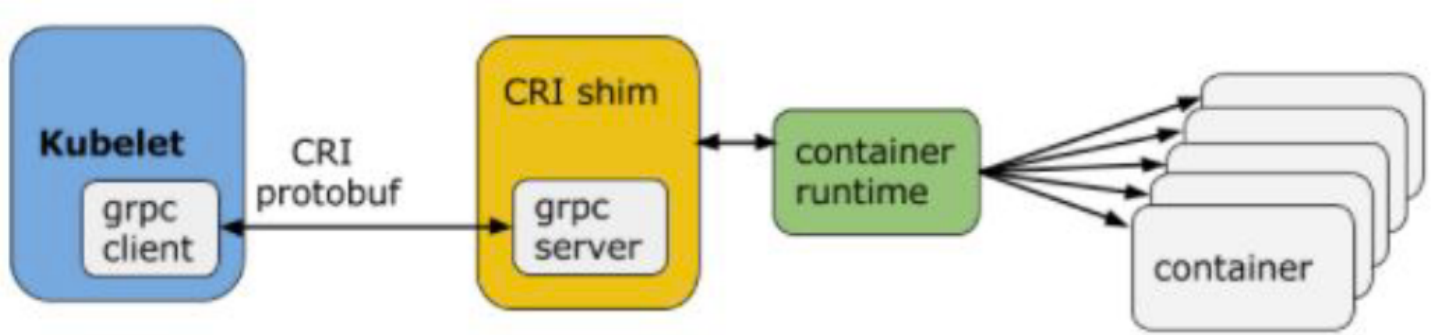
Standardization became a fact



@jorgemoralespou

What is Container Runtime Interface (CRI)?

- A gRPC interface and a group of libraries
- Enables Kubernetes to use a wide variety of container runtimes
- Introduced in Kubernetes 1.5
- GA in Kubernetes 1.7



CRI Implementations

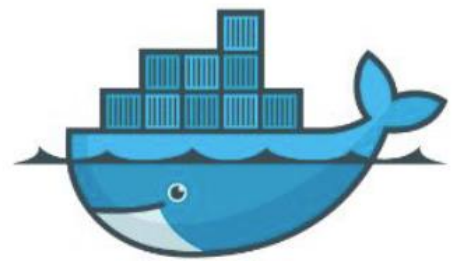
cri-o

cri-containerd

rktlet

frakti

virtlet



dockershim

Why Use Container Orchestrators?

Though we can argue that containers at scale can be maintained manually, or with the help of some scripts, container orchestrators can make things easy for operators.

Container orchestrators can:

- Bring multiple hosts together and make them part of a **cluster**
- **Schedule** containers to run on different hosts
- Help containers running on one host **reach out to containers running on other hosts** in the cluster
- Bind containers and storage
- Bind containers of similar type to a higher-level construct, like **services**, so we don't have to deal with individual containers
- Keep resource usage in-check, and optimize it when necessary
- Allow secure access to applications running inside containers.

With all these built-in benefits, **it makes sense to use container orchestrators to manage containers**.

Basic things we can ask Kubernetes to do

- Start 5 containers using image atseashop/api:v1.3
- Place an internal load balancer in front of these containers
- Start 10 containers using image atseashop/webfront:v1.3
- Place a public load balancer in front of these containers
- It's Black Friday (or Christmas), traffic spikes, grow our cluster and add containers
- New release! Replace my containers with the new image atseashop/webfront:v1.4
- Keep processing requests during the upgrade; update my containers one at a time

Other things that Kubernetes can do for us

- Basic autoscaling
- Blue/green deployment, canary deployment
- Long running services, but also batch (one-off) jobs
- Overcommit our cluster and evict low-priority jobs
- Run services with stateful data (databases etc.)
- Fine-grained access control defining what can be done by whom on which resources
- Integrating third party services (service catalog)
- Automating complex tasks (operators)

Declarative vs imperative

- Our container orchestrator puts a very strong emphasis on being declarative
- Declarative:
  - *I would like a cup of tea.*
- Imperative:
  - *Boil some water. Pour it in a teapot. Add tea leaves. Steep for a while. Serve in cup.*
- Declarative seems simpler at first ...
- **... As long as you know how to brew tea**

Declarative vs imperative in Kubernetes

- Virtually everything we create in Kubernetes is created from a spec
- Watch for the spec fields in the YAML files later!
- The spec describes how we want the thing to be
- Kubernetes will reconcile the current state with the spec (technically, this is done by a number of controllers)
- When we want to change some resource, we update the spec (desired state)
- Kubernetes will then converge that resource

Kubernetes network model

- TL,DR:

  Our cluster (nodes and pods) is one big flat IP network.

- In detail:
  - all nodes must be able to reach each other, without NAT
  - all pods must be able to reach each other, without NAT
  - pods and nodes must be able to reach each other, without NAT
  - each pod is aware of its IP address (no NAT)
  - Kubernetes doesn't mandate any particular implementation

Kubernetes network model: the good

- Everything can reach everything
- No address translation
- No port translation
- No new protocol
- Pods cannot move from a node to another and keep their IP address
- IP addresses don't have to be "portable" from a node to another (We can use e.g. a subnet per node and use a simple routed topology)
- The specification is simple enough to allow many various implementations

Kubernetes network model: the less good

- Everything can reach everything
    - if you want security, you need to add network policies
    - the network implementation that you use needs to support them
- There are literally dozens of implementations out there (15 are listed in the Kubernetes documentation)
- It looks like you have a level 3 network, but it's only level 4 (The spec requires UDP and TCP, but not port ranges or arbitrary IP packets)
- *kube-proxy* is on the data path when connecting to a pod or container, and it's not particularly fast (relies on userland proxying or iptables)

Kubernetes network model: *kube-proxy*

- Don't worry about the warning about *kube-proxy* performance
- Unless you:
    - routinely saturate 10G network interfaces
    - count packet rates in millions per second
    - run high-traffic VOIP or gaming platforms
    - do weird things that involve millions of simultaneous connections (in which case you're already familiar with kernel tuning)

Conclusions

Thoughts from one of the creators of Kubernetes

Joe Beda @jbeda · May 8

First off: Kubernetes *is* a complex system. It does a lot and brings new abstractions. Those abstractions aren't always justified for all problems. I'm sure that there are plenty of people using Kubernetes that could get by with something simpler. /3

Joe Beda @jbeda · May 8

When you create a complex deployment system with Jenkins, Bash, Puppet/Chef /Salt/Ansible, AWS, Terraform, etc. you end up with a unique brand of complexity that *you* are comfortable with. It grew organically so it doesn't feel complex. /6

Joe Beda @jbeda · May 8

This is a place where, IMO, Kubernetes adds value. Kubernetes provides a set of abstractions that solve a common set of problems. As people build understanding and skills around those problems they are more productive in more situations. /8

Abstractions are NOT a new thing,

It's how CS evolved

BUT…

Architecture Astronauts

> When you go too far up, abstraction-wise, you run out of oxygen. Sometimes smart thinkers just don't know when to stop, and they create these absurd, all-encompassing, high-level pictures of the universe that are all good and fine, but don't actually mean anything at all.

2001, Joel Spolsky
Co-founder @stackoverflow @trello

RFC 1925

1 April 1996

“

(6) It is easier to move a problem around (for example, by moving the problem to a different part of the overall network architecture) than it is to solve it.

(6a) (corollary). It is always possible to add another level of indirection.

”

FTSE

Fundamental Theorem of Software Engineering

> We can solve any problem by introducing an extra level of indirection.

The theorem is often expanded by the humorous clause

"…except for the problem of too many levels of indirection"

too many

Abstractions = **Complexity**

may create intrinsic

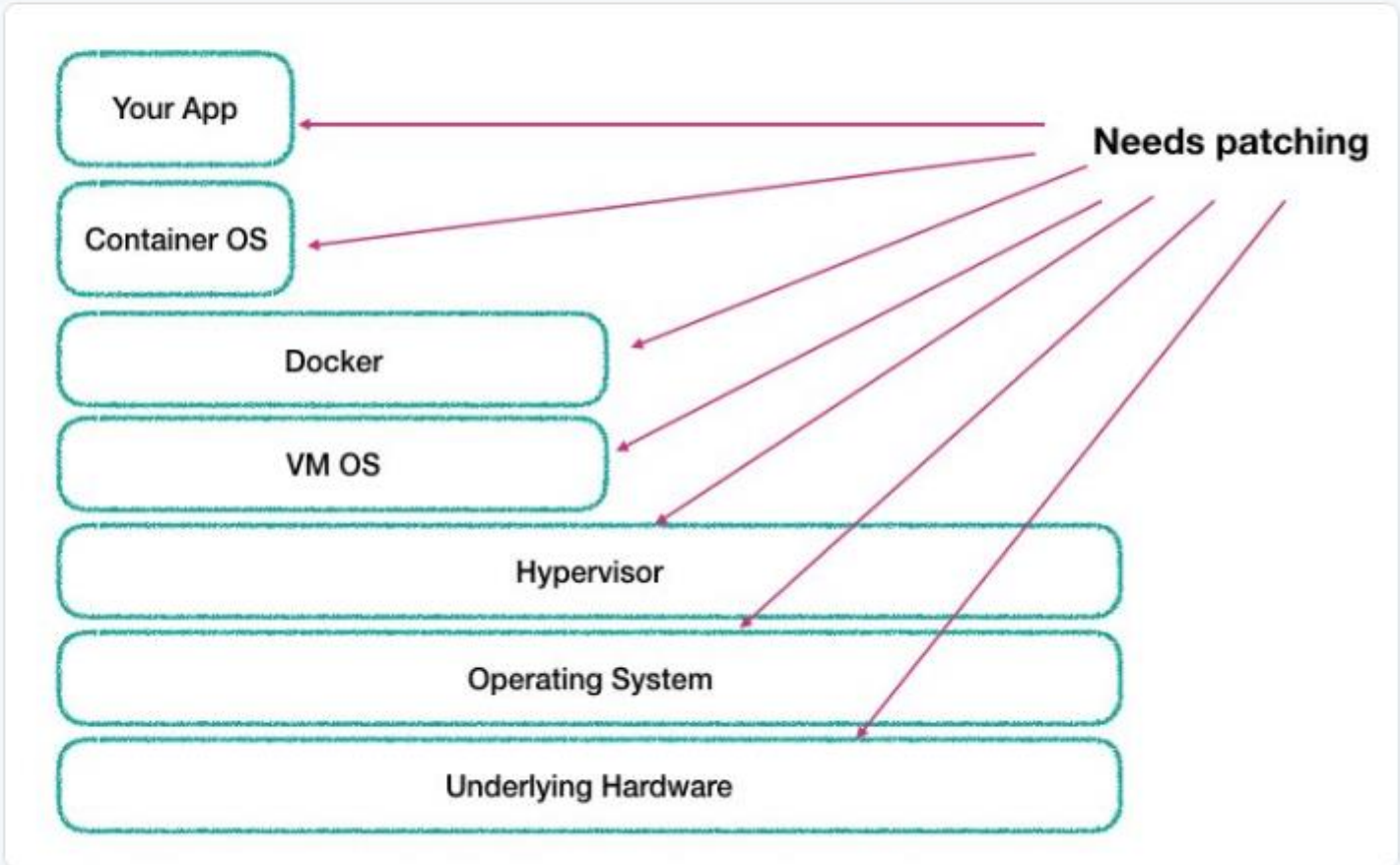
issues of their own

Kubernetes on Openstack

Pinned Tweet

Sam Newman @samnewman · Jan 14

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?



56 1.1K 1.7K

Oh, you want to use ML on K8s?

First, become an expert in:

- Containers
- Packaging
- Kubernetes service endpoints
- Persistent volumes
- Scaling
- Immutable deployments
- GPUs, Drivers & the GPL
- Cloud APIs
- DevOps
- ...

That’s all…

Links

- https://loige.co/from-bare-metal-to-serverless/
- https://hackernoon.com/why-the-fuss-about-serverless-4370b1596da0
- https://www.martinfowler.com/articles/serverless.html
- https://www.slideshare.net/randybias/the-history-of-pets-vs-cattle-and-using-it-properly
- https://www.slideshare.net/spnewman/what-is-this-cloud-native-thing-anyway
- https://www.slideshare.net/spnewman/confusion-in-the-land-of-the-serverless
- https://www.slideshare.net/JorgeMorales124/build-and-run-applications-in-a-dockerless-kubernetes-world
- https://medium.com/@adriaandejonge/moving-from-docker-to-rkt-310dc9aec938
- https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html
- https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon