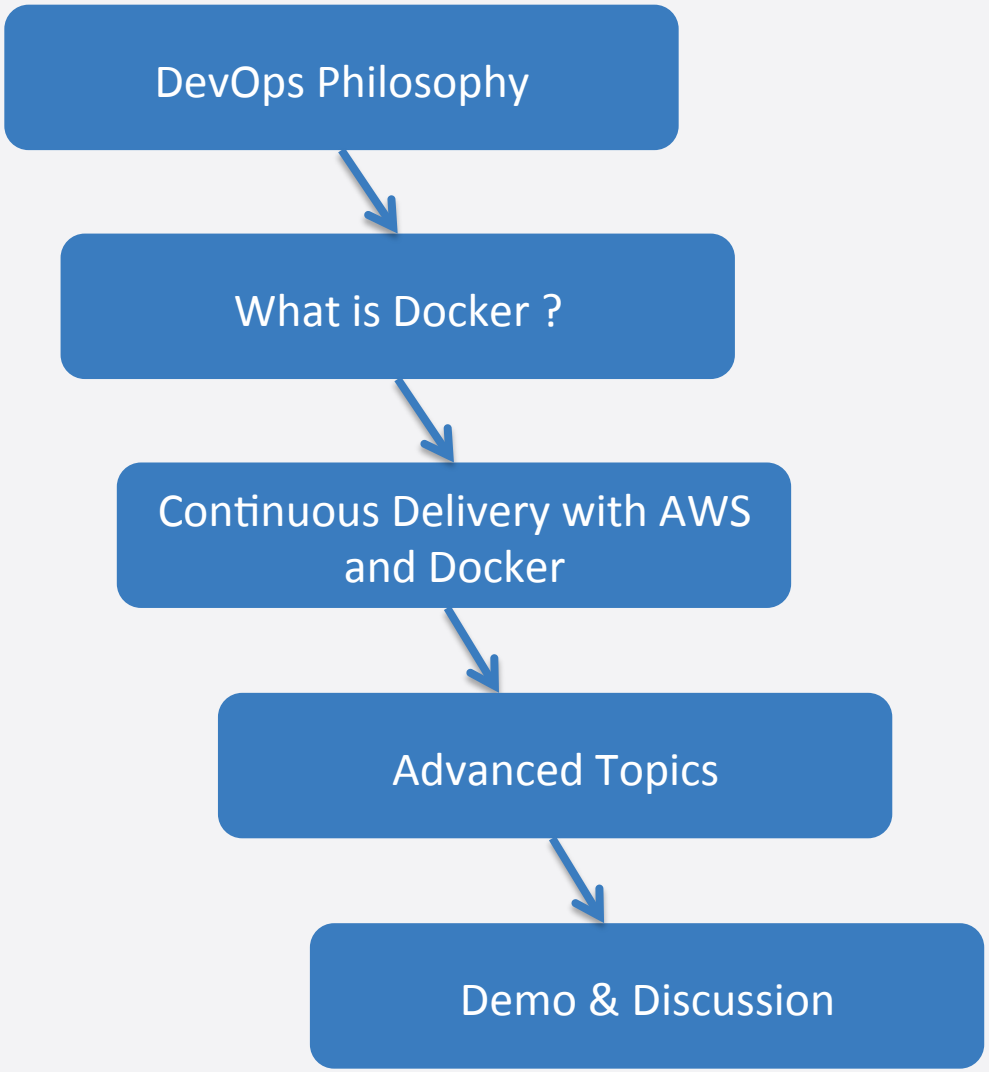

Micro Services and Continuous Deployment with Docker
Andrew Aslinger
Dec 4th 2014
OpenWhere™
OpenWhere

Agenda


DevOps Philosophy
What is Docker ?
Continuous Delivery with AWS and Docker
Advanced Topics
Demo & Discussion

Who am I?

Senior Cloud Architect for OpenWhere

Passion for UX, Big Data, and Cloud/DevOps

Previously Designed and Implemented automated DevOps processes for several organizations

Multiple environments, many user / developer groups, big data clusters, hundreds of machines

A Recovering “Chef”

Certified AWS Architect

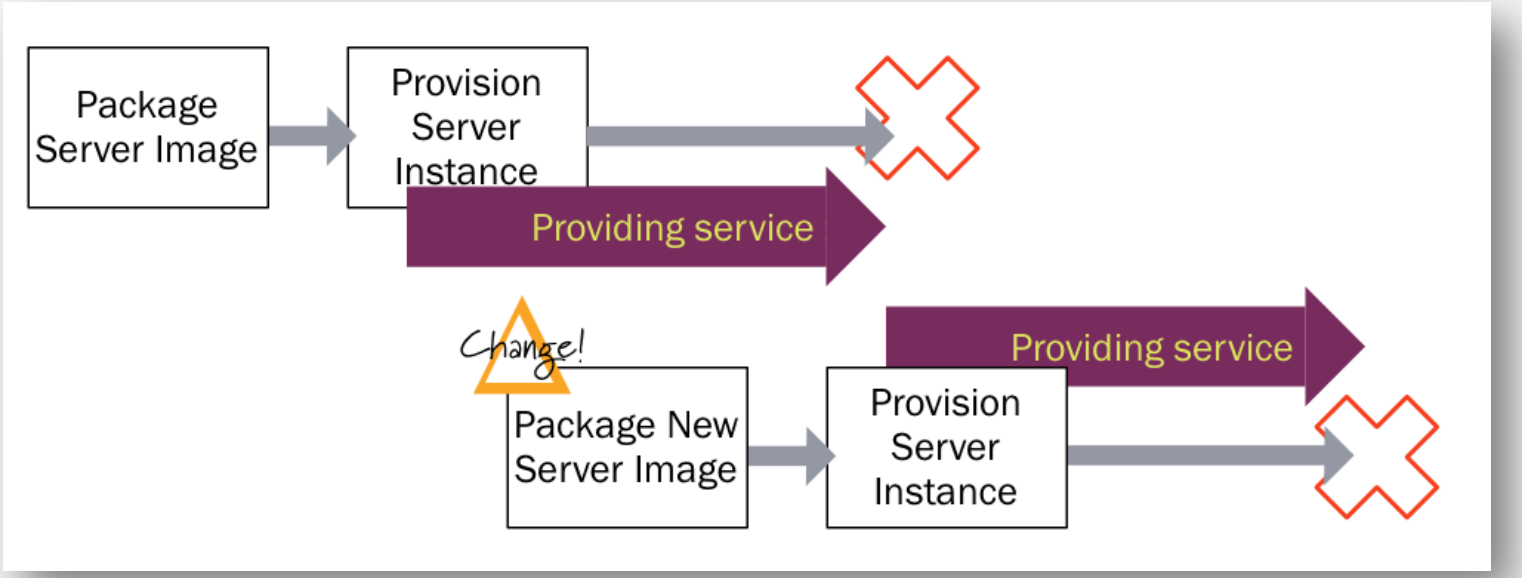
DevOps Principles

OpenWhere DevOps Philosophy:
Apply Software principles to infrastructure

1. Immutable Infrastructure
2. Purpose Built Environments
3. Self service for the entire Landscape
4. Create Programmatic Bill of Material
5. Simplify the tool chain & process
6. DevOps code should co-exist with application code
7. Architect, don't Broker, for Cross Cloud deployments
8. Auto-scale from start
9. Build in cost optimization

1. Immutable Infrastructure

- Virtual Infrastructure components are replaced for every deployment, rather than being updated
- Focus on one process (build) vs. two (build & maintain)



http://martinfowler.com/bliki/ImmutableServer.html

2. Use Purpose Built Environments

Fixed, Static Environments

- Average 50% variance between production & lower environments
- Supporting environments have variable demand, low overall utilization, & minimal support

Fixed Environments	
Development	Staging
Integration	Demonstration
Test /QA	Training
Production	Etc.

Dynamic, on-demand environments

- Built and scaled for specific purposes
- Exist only for the task duration

Dynamic Environments
Development for Sprint 11 (3 weeks)
User Test for User Story US 217 (14 hours)
Regression Test for Defect 42 (1 hour)
Performance Test for release 2.3.1 (4 hours)
Training for release 2.3.2 (8 hours/day)
Production for release 2.3.0 (1 month)

3. Self Service for the entire Landscape (systems not servers)

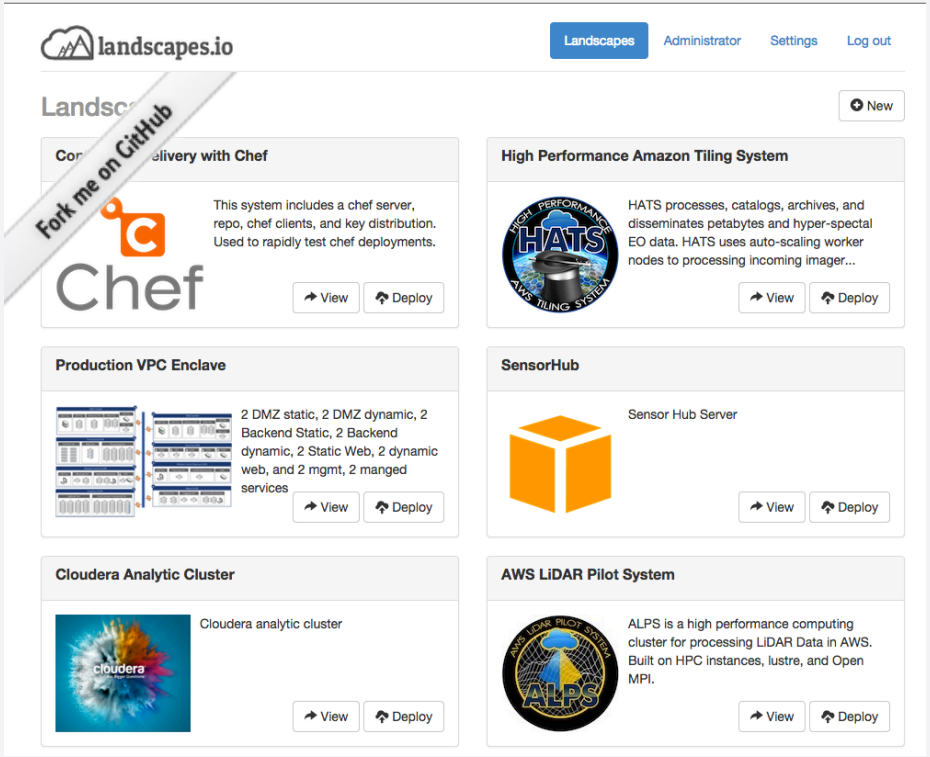
- Single server is not a viable unit of work

  "In today's distributed compute environment, developers can't develop on local workstations."

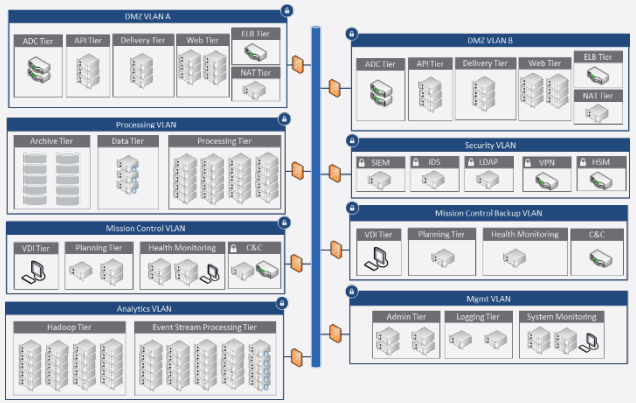
- Teams want self-service to full systems

  "I want a LIDAR processing system ***not*** 12 servers, 2 subnets, database, NAT, load balancer, etc"

- Shift self-service focus from servers and components to applications & systems (landscapes)



4. Create Programmatic Bill of Material

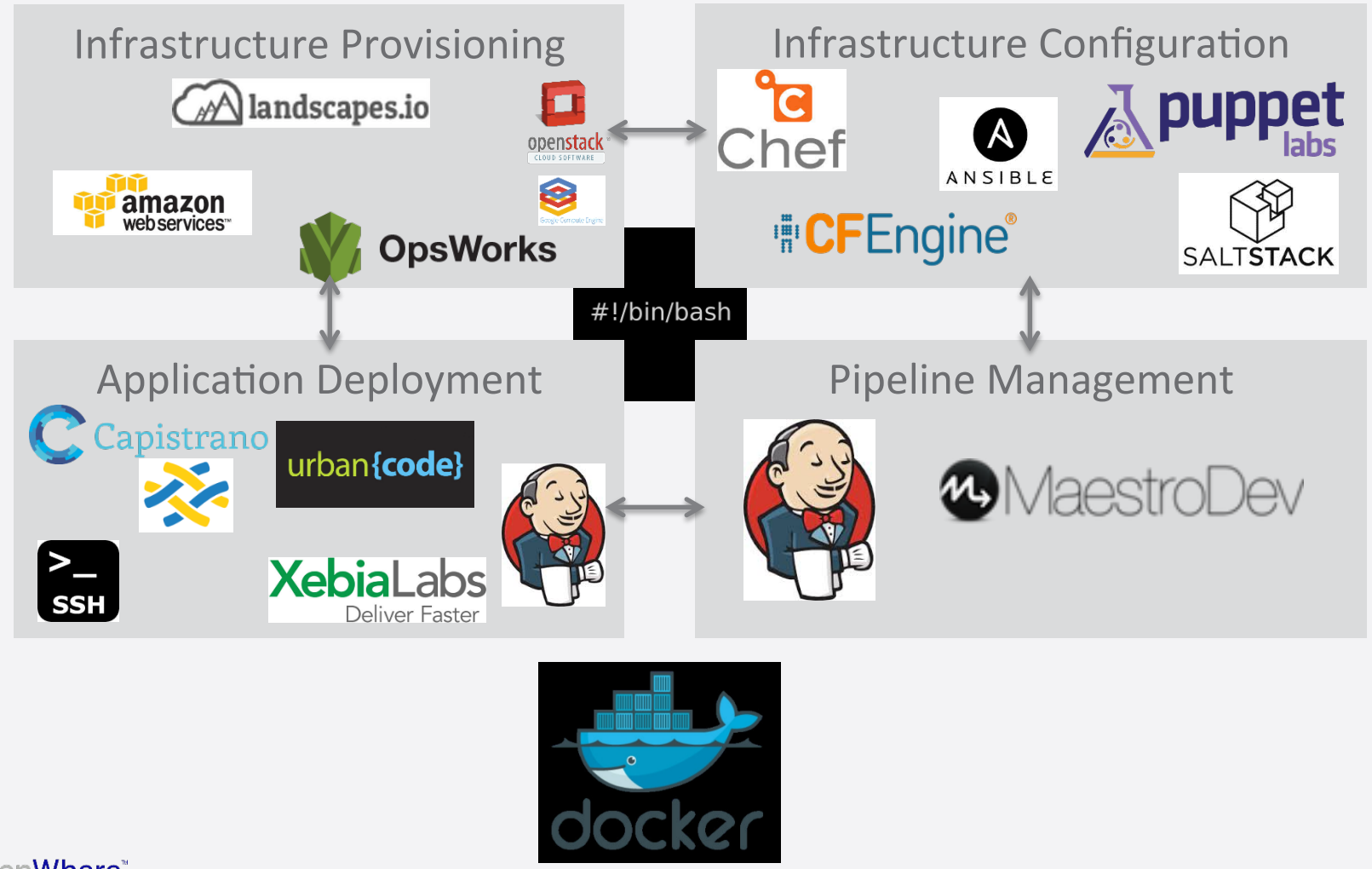



```
{ "AWSTemplateFormatVersion“:"2010-09-09",
  "Description": "Creates a VPC with 8 Subnets: 2 DMZ, 2 Backend, 2
Web, and 2 mgmt ",
  "Parameters": {
     "VpcName": {
        "Description": "Should be a unqiue name of the VPC",
        "Type": "String“
   }, "KeyName": {
        "Description": "Name of Key for NAT instances",
        "Type": "String",
        "Default“:"OpenWhereRoot“
   },"CidrBlock":{
        "AllowedPattern":
"(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})/(\\d{1,2
...
```


The entire system is captured as a software code. This allows the infrastructure to be version controlled and replicated like any other software asset.

- **Build infrastructure from JSON Templates**
  - JSON description file (programmatic Bill of Materials)
  - Perform programattic dependency and impact analysis - just like on code
  - Used to initialize Docker Containers
- **Resource specific templates**
  - AWS Cloud Formation
  - OpenStack HEAT
  - Docker File / Fig
  - OpenWhere has developed a Javascript DSL to generate these cloud specific templates

A very, very small sample of the tool space . . .



OpenWhere DevOps Philosophy:
Apply Software principles to infrastructure

1. Immutable Infrastructure
2. Purpose Built Environments
3. Self service for the entire Landscape
4. Create Programmatic Bill of Material
5. Simplify the tool chain & process
6. **DevOps code should co-exist with application code**
7. **Architect, don't Broker, for Cross Cloud deployments**
8. **Auto-scale from start**
9. **Build in cost optimization**

Docker helps enable many of these principles

What is Docker?

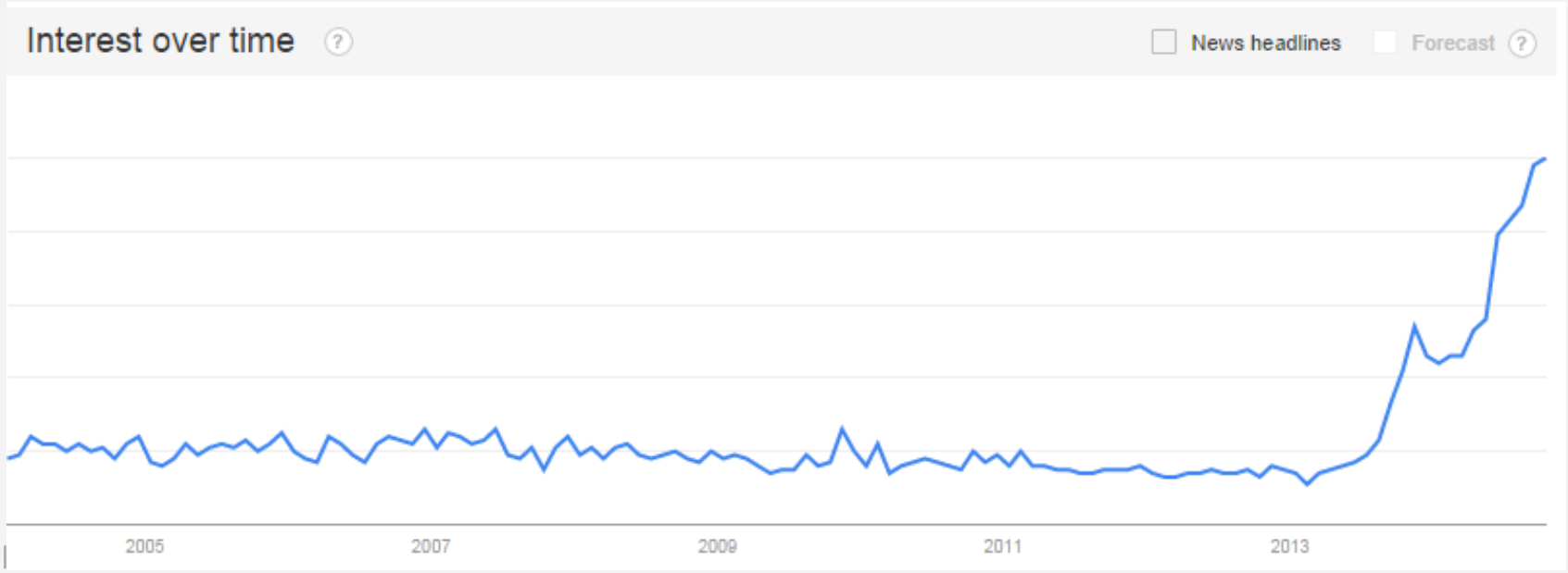
What is Docker?

"Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications"

Yet another DevOps tool

light weight container

Latest hype?



What is Docker?

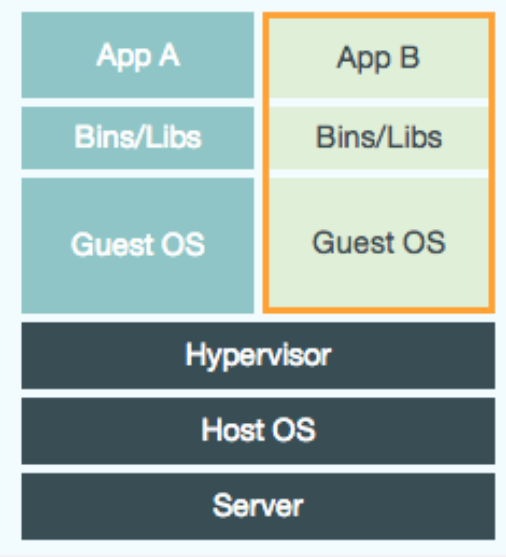
Docker really does for any unit of work and all it's dependencies (OS, configuration, binaries, any software language etc.) what the JVM did for your code

- ***Have something in Python or Fortran*** *? – Sure*
- ***Gnarly C Libraries –*** *All right! How about I run it on your Mac and EC2*
- ***Conflicting dependencies across two services*** *– not a problem!*

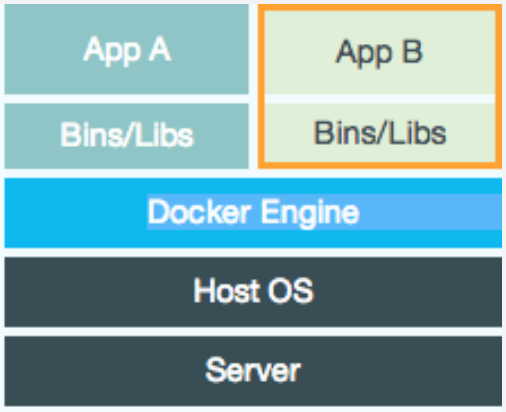
Docker Benefits

- **Portability Across Machines**
  - Local Vagrant like deployment, Remote Deployment including Amazon EC2
  - Independent of host OS
  - Lightweight vs VMs
- **Supports DevOps Best Practices & Re-use**
  - Infrastructure as Code
  - Container Extensibility
- **Fits into a Continuous Deployment Pipeline**
- **Isolation and Single Responsibility**
  - Reduce dependency conflicts, security
- **Stick any unit of work in a black box with its dependencies**
  - Code with complex dependencies and configuration, short lived tasks, long running services, etc.

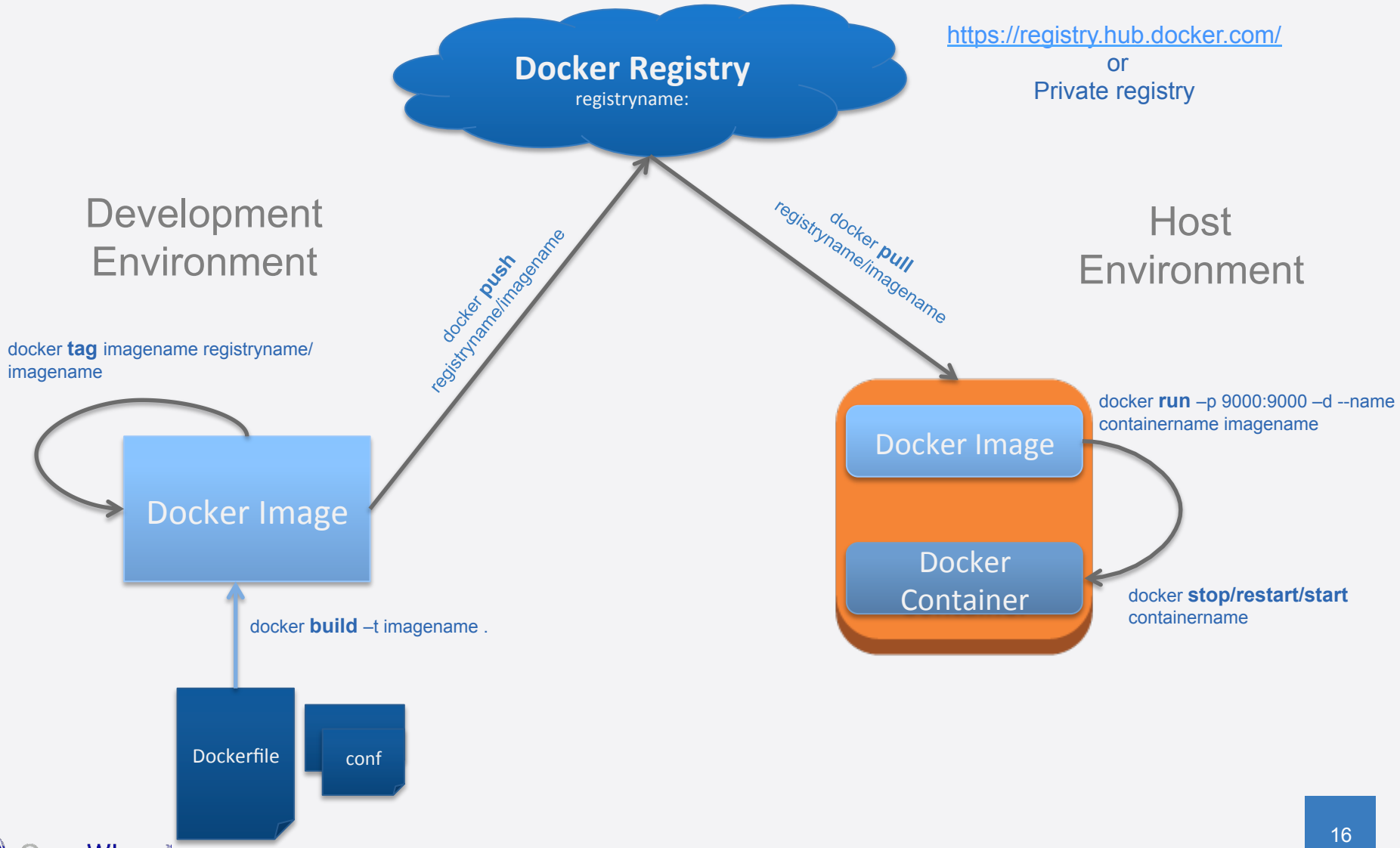
Virtual Machines



Docker



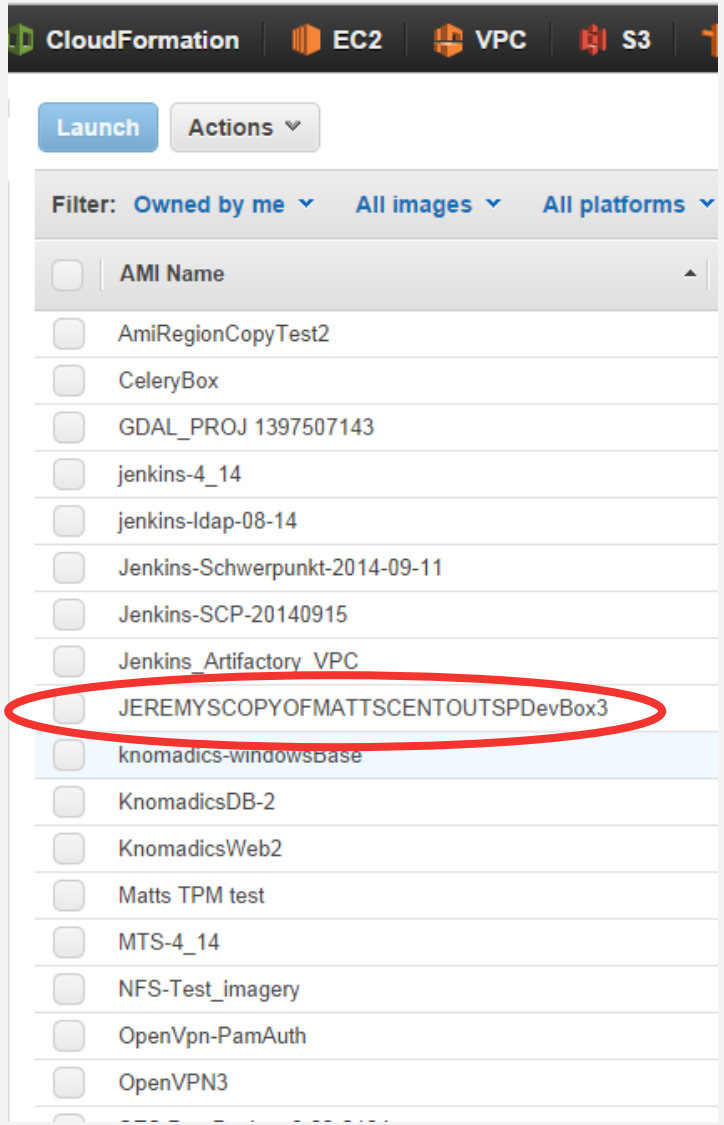
A Typical Docker Flow with six simple commands



Docker with AWS

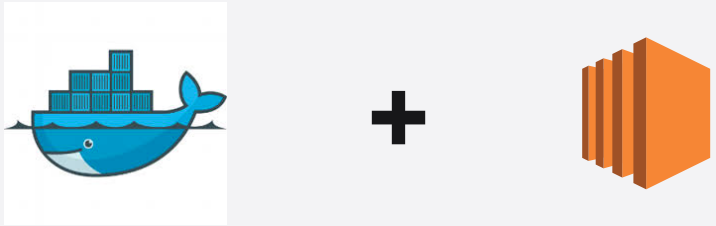
Why Docker with EC2?

- **Build once, run many places**
  - Dev Machines, different clouds, different VPCs, share and build
  - Not tied to box size or guest OS
- **AMI's are difficult to rebuild and more difficult to maintain**
  - Snapshots retain baggage
  - There is no "git" version control for AMI
  - AMI's are not cross region compatible
  - Other AMI tools like Amiator aren't easy to use
- **Easier to achieve immutable infrastructure and one click deploys**
- **Treat your infrastructure as code with a Continuous Deployment Pipeline**

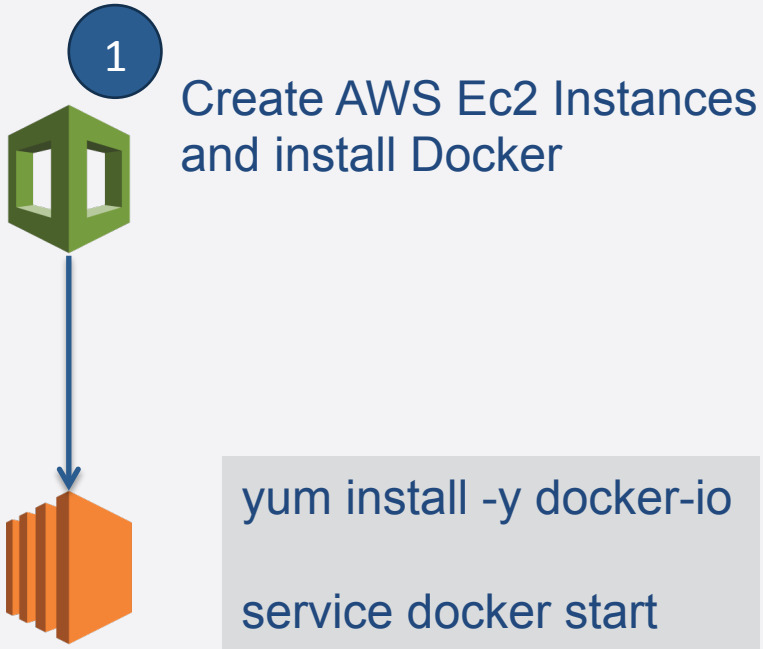


Why Docker with EC2?

- Separate your data and state from your image
- Machine layers evolve at different velocity and have different reuse (Example Later)
  - Core OS
  - Organization Level Requirements
  - Application Level Requirements
  - Application Code
- Test and version control your deploy artifacts and configuration together
  - Same container for developers and EC2
  - Java Example
- Stop maintaining AMIs
- HPC algorithm distribution unit (Apache Mesos)



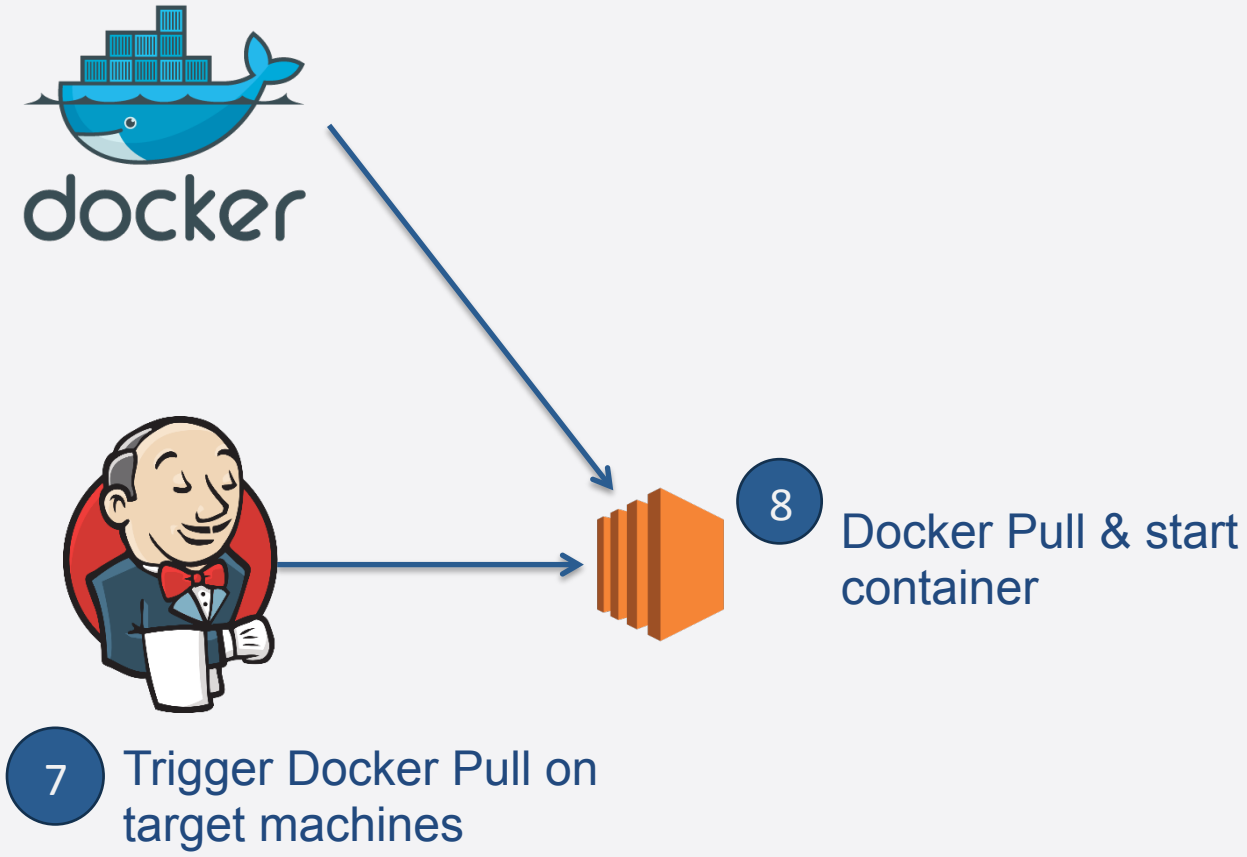
Docker Continuous Deployment Flow:
Set up AWS infrastructure



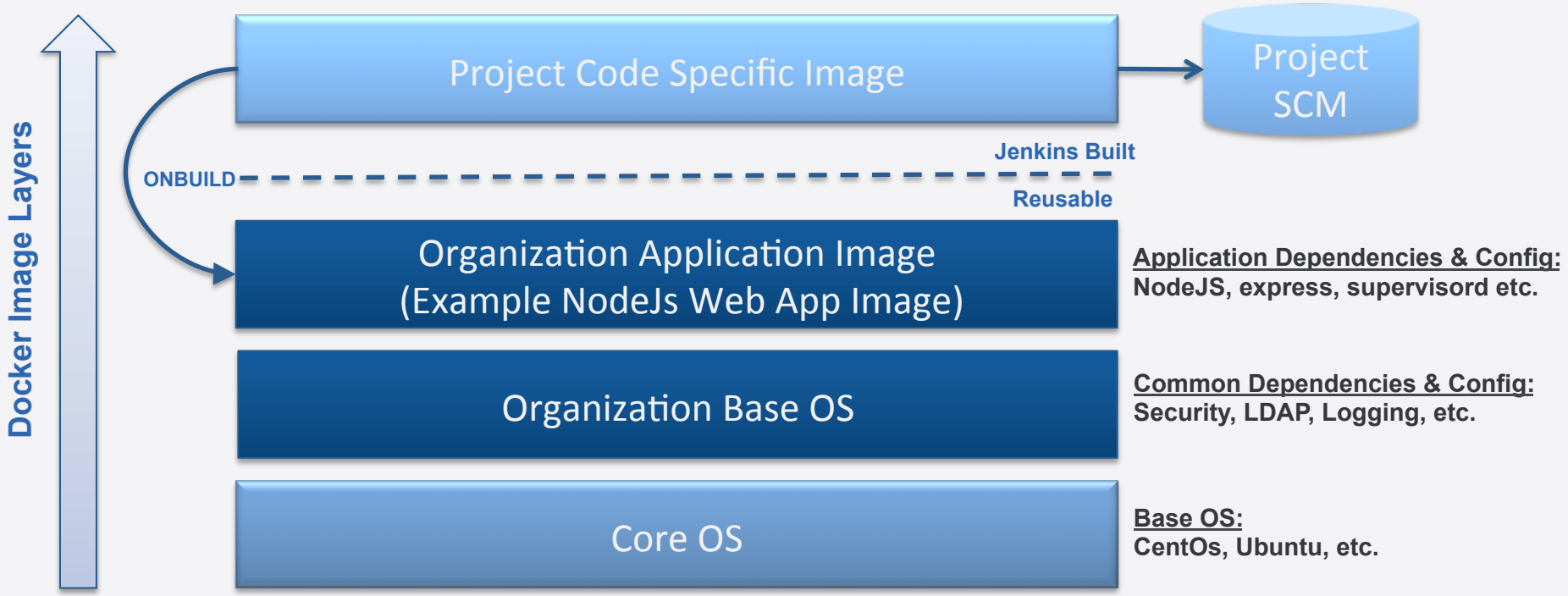
Docker Continuous Deployment Flow:

Build and push docker image with latest code

Docker Continuous Deployment Flow:
Trigger EC2 to pull latest Docker image



Advanced #1: Docker Image Layers using FROM and OnBuild commands in the Docker File



Machine layers can evolve at different velocity and have different levels of reuse

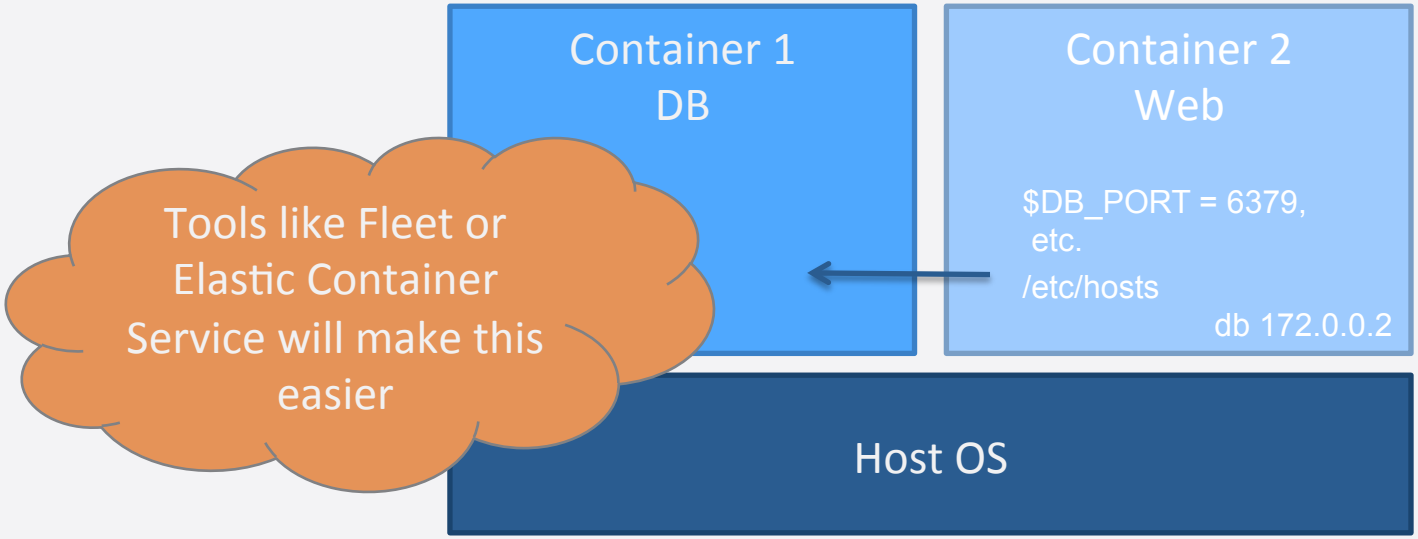
Advanced #2: Linked Containers

Run multiple containers on a machine. Each with a responsibility linked by ports / address

https://docs.docker.com/userguide/dockerlinks/

--link name:alias

docker run -d -P --name web --link db:db training/webapp python app.py

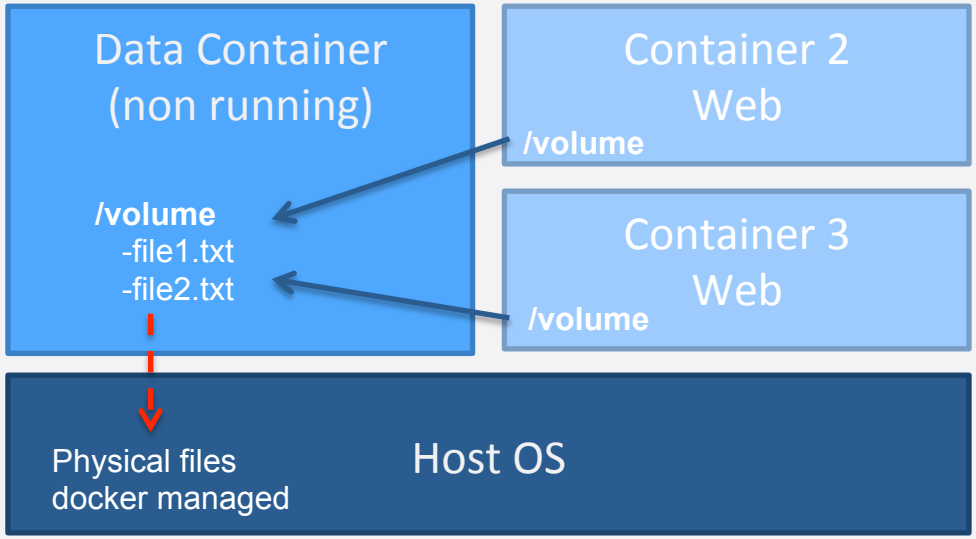


Advanced #3: Data Containers

A container which allows sharing of data across containers
https://docs.docker.com/userguide/dockervolumes/

Benefits: Change image independent of data (maintainability)
Enables Data sharing patterns (single responsibility)
Limitations: Data is Tied to Host OS

sudo docker run -d --volumes-from dbdata --name db2 training/postgres



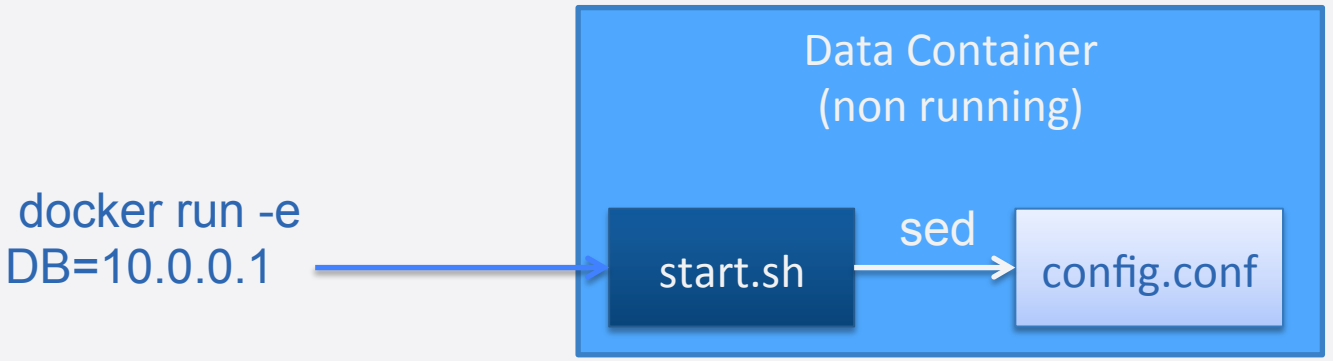
Advanced #4: Customize on Init

You need to parameterize and modify items in a container based on the environment, and ENV variables won't work

Examples: Set a IP address or DNS name,
Set a DB password

Pattern:

1. **Load a script onto the box as part of an building the image**
2. **Run script at container launch**
3. **Script uses environment variables to customize settings files etc**
4. **Environment variables are passed to the docker run Command**



Service Discovery & Registration is an emerging area

Service Discovery

Service Discovery Needs / Drivers

- Managing large amounts of Micro-Services quickly becomes unmanageable
- Traditional Approaches of orchestration often require static endpoints which do not scale and are more difficult to manage
- Docker introduces patterns of linking containers but one quickly runs into the need to link services across hosts
- Solution must deal with Cloud Focused Architectures, Rolling Deploys, Multiple Environments, and Elasticity, etc.

http://www.cargolaw.com/2007nightmare_ital.florida.html

Ideal Solution Goals

- "Services just find each other and work"
  - Auto-wiring / Full Automation
    - Services automatically discover one another
    - Works with machines coming/going and autoscale groups
    - Minimal or no host hardcoding
    - Easy to manage
    - Supports continuous deployment
  - Scalability to hundreds/thousands of services
  - Easy to use
  - Minimal impact / code changes to existing services
  - **Docker / Container Process Integration**
  - Versatility & Host and Environment Agnostic
  - Health Checking
  - Supports Orchestration of Deployments
  - Upward trajectory for tech and room for growth

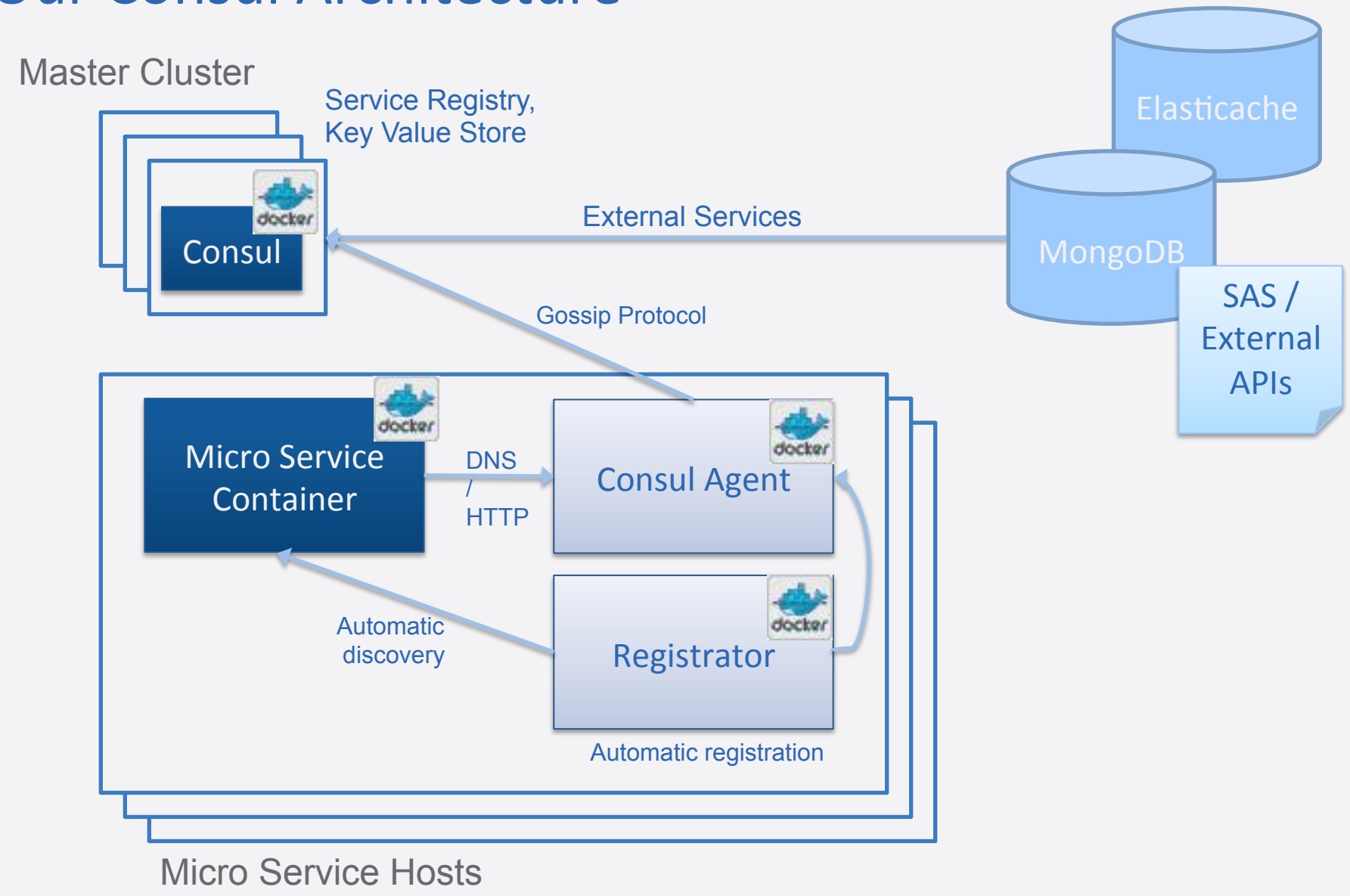
Comparison Matrix

Stack	Smart Stack	Consul	Etcd	CoreOS	AWS Manual
Techs	Nerve, Synapse, HAProxy, Zookeeper	consul	etcd	CoreOS, fleet, etcd	Route53 + ELB + CloudFormation
Link	http://nerds.airbnb.com/smartstack-service-discovery-cloud/	http://www.consul.io	https://github.com/coreos/etcd	https://coreos.com/	Custom AWS Solution
Pros	• Some people use with docker • Well defined	• Opinionated • Multi-data center • Key / value • Strong health checking • DNS + Rest • Gossip built on serf protocol • UI	• Simple / Fast (Raft based) • Key/value • Option for DNS via SkyDNS2	• Docker centric • Security focus • Orchestration of Containers	• Simple and easy • Known approach
Cons	• More complex stack • Zookeeper underpinnings, no dynamic fleets	• Docker is added on not centric like CoreOS	• Manual solution when used alone • Less Opinionated	• Requires their OS (less proven off cloud)	• Requires hardcoding • Doesn't scale or translate outside AWS

Selection

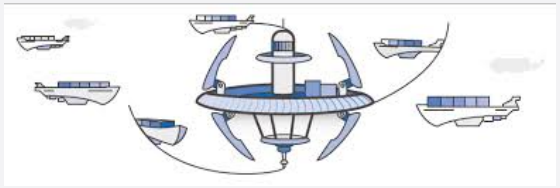
- Selection: Consul + Docker Integrations
  - Consul + some external open source Docker integrations met all the goal criteria except orchestration
    - Orchestration easy to accomplish with current Cloud Formation Process
    - Elastic Container service also will provide orchestration
- Runners Up
  - CoreOS close but concerns over on-premise deploys. Also lacked easy DNS and was a more drastic change as requires different host OS
  - Etcd by itself and even with docker integrations was still too manual and didn't require enough flexibility
  - AWS while known was more short term as offered less flexibility and lacked other features including non-http health checks, etc.

Our Consul Architecture



Orchestration

- **How do I manage deployment, resource allocation, linking, etc. of all these containers?**
  - Container as a first class citizen instead of the VM
  - If I split things smaller, I want more automation in the management
  - Docker provides common patterns and interface of work (start / stop etc.)
  - Manage the container linking and items not in my Dockerfile
- **Options:**
  - Mesos:
    - **Launch tasks that contain docker containers**
    - **http://mesos.apache.org/documentation/latest/docker-containerizer/**
  - AWS Elastic Container Service (Preview):
    - **Dynamically allocate containers across a cluster**
    - **Ground up built for minimal thought Docker in the cloud**
    - **http://aws.amazon.com/ecs/**
  - Google Container Engine
    - **Manage Docker containers on Google Compute Engine**
    - **https://cloud.google.com/container-engine/**



OpenWhere Docker Wins

1. **Full continuous deployment and write once, deploy many times realized**

2. **Developer Replacement for vagrant**
   - Less complexity
   - Test with what you deploy!

3. **Deployment build process co-exists with the software!**
   - Reduced Maintenance, developer buy-in, testability

4. **Awesome Re-use with minimal code**
   - 1 line Dockerfile(s) for new MEAN Stack apps

5. **Simplicity (Streamlined DevOps)**

OpenWhere Docker Burns

- **Learning Curve / Maturity**
  - Learning and growing to a Docker way for each DevOps task (still in progress)
  - **Example**: Need "tricks" to do things like have more than one process running in a container
  - **Example:** Learning layers are independent and don't retain state between RUN commands
- **A Docker Container is not the same as a typical virtual machine**
  - Basic stuff like syslog off and tar, wget not installed
  - Base Image: http://phusion.github.io/baseimage-docker/
  - Ideal State: Single process vs. Single Responsibility
- **Containers not restarting on EC2 Restart**
  - We hacked a cloud-init work around
- **Updating to new images is dirty**
  - Manually clean up images / containers or run out of disk!
- **Debugging can be more complex:**
  - **Largely fixed In Docker 1.3: `docker exec –I –t [container-name] bash`**

Recommendations & Next Steps

- Docker works great for a Continuous Deployment flow on EC2
  - Especially for stateless applications or a micro-service architecture

- Complex cluster deployments, stateful applications or databases, are harder to achieve
  - You may need to roll your own solution or integrate with another technology

- Docker Next Steps: Programs and IR&D
  - **Security** Testing & Recommendations
  - **Service Discovery** (Consul, Fleet, Kubernetes, Mesos, Helios, Centurion, libswarm, etc.)
  - **Data Persistent Patterns** (Data Lake, Shared Volumes, etc.)

Questions

Contact Information

Andrew Aslinger
Senior Systems Architect
OpenWhere
aaslinger@openwhere.com
@aaa572
http://owaaa.github.io/

Andrew Heifetz
Chief Cloud Officer
OpenWhere
aheifetz@openwhere.com
@andyheifetz
http://www.linkedin.com/pub/andrew-heifetz
Cell: 240-481-7442