# Containers at Scale – Kubernetes and Docker

Michael Heldebrant
Cloud Infrastructure Solutions Architect
Red Hat
January 2015

# AGENDA

- What is Kubernetes
- Why Kubernetes?
- Kubernetes concepts
- Kubernetes networking
- Patch cycle with Kubernetes
- OpenShift 3.0 roadmap

# Kubernetes?

## Hortator (inciter; encourager, exhorter; urger):

# Kubernetes:

κυβερνήτης: *Greek for “pilot” or “helmsman of a ship”*

the open source cluster manager from Google

# What is Kubernetes

- June 2014 Google open sourced a container management project
- Google has been using containers for over a decade
- Red Hat is collaborating in the kubernetes project

http://www.redhat.com/en/about/blog/red-hat-and-google-collaborate-kubernetes-manage-docker-containers-scale

# Why Kubernetes?

Docker is an engine, container and image format with limited networking between hosts.*

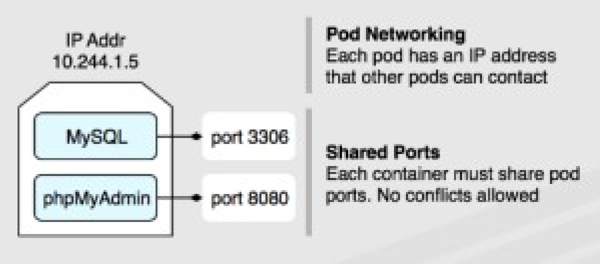
Kubernetes is a way to:

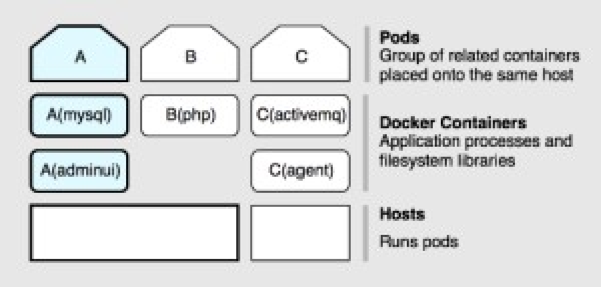
- describe and launch
- monitor state and maintain, increase or reduce copies of containers
- container oriented networking for non kubernetes native applications

Kubernetes builds on Docker to make management of many containers like managing containers on a single system.

*prior to recent Docker roadmap annoucements

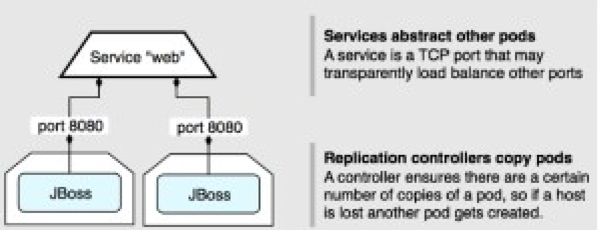
# Kubernetes Concepts


IP Addr
10.244.1.5
MySQL
phpMyAdmin
port 3306
port 8080
Pod Networking
Each pod has an IP address that other pods can contact
Shared Ports
Each container must share pod ports. No conflicts allowed


A
B
C
A(mysql)
B(php)
C(activemq)
A(adminui)
C(agent)
Pods
Group of related containers placed onto the same host
Docker Containers
Application processes and filesystem libraries
Hosts
Runs pods

- Pods
  - Collection of co-located containers with a unique ip address
  - Connect containers in the pod to each other via localhost networking
  - Shared volume(s)
  - Labels for Replication Controllers and Services to select

# Kubernetes Concepts



**Services abstract other pods**
A service is a TCP port that may transparently load balance other ports

**Replication controllers copy pods**
A controller ensures there are a certain number of copies of a pod, so if a host is lost another pod gets created.

- Replication Controllers
  - Keep N copies of a pod running or update N
  - Pod templates describe the pod to manage
- Services
  - Stable IP and ports for connecting pods together across a cluster of container hosts
  - Services are long lived compared to Pods

# Kubernetes Concepts

- Labels
  - Key Value pairs attached to primitives (pods, rep. controllers, services)
  - Labels are not meant to be unique
  - Labels are used by replication controllers and services to match pods
    - key1
    - key1 = value11
    - key1 != value11
    - key1 in (value11, value12, ...)
    - key1 not in (value11, value12, ...)
  - Use multiple key-value pairs to cross cut the set to select

# Kubernetes Architecture

- Master
  - kube-apiserver – interface between users and kubernetes
  - kube-controller-manager – monitors replication controllers and adds/removes pods to reach desired state
  - kube-scheduler – schedules pods to minions
  - etcd – key value store over HTTP
- Minions
  - kublet – node level pod management
  - proxy – forward traffic to pods
  - cadvisor – resource monitor
  - docker – engine for containers

# Kubernetes Supporting Infrastructure

- Docker images
  - Docker factory to build images
- Docker image management
  - Docker-registry
- Load balancers
  - HA-proxy or equivalent to route traffic external to the cluster to minions

# 5 minute Docker crash course

- Docker build host
  - yum --enablerepo rhel-7-server-extras-rpms install docker
  - add –insecure-registry=**registry-host**:5000 to /etc/sysconfig/docker OPTIONS line
  - systemctl start docker
- Docker images
  - Red Hat base os images: https://access.redhat.com/search/browse/container-images
  - docker load -i rhel-server-docker-7.0-*.tar.gz
  - docker tag rhel-server-docker-7.0-23.x86_64 rhel7
- Create Dockerfile to build and run a simple application (apache httpd)

```
FROM rhel7

MAINTAINER Michael Heldebrant mheldebr@redhat.com

# Install httpd

RUN yum -y install httpd && yum clean all

# Set up apache to run

ENTRYPOINT ["/usr/sbin/httpd", "-DFOREGROUND"]
```

# 5 minute Docker crash course

- Docker registry for testing
  - docker run -d -p 5000:5000 -e STORAGE=local -e STORAGE_PATH=/tmp/ --name=registry registry
- Build images
  - cd dockerfiledirectory
  - docker build -t registry-host:5000/httpd:latest .
- Push images
  - docker push registry-host:5000/httpd:latest

# 10 minute Kubernetes install

- https://access.redhat.com/articles/1198103#start
  - Install on master and minions

    -
      ```
      yum --enablerepo rhel-atomic-host-beta-rpms --enablerepo rhel-7-server-extras-rpms install kubernetes
      ```
  - Edit the config files in /etc/kubernetes and start services on master

    -
      ```
      for SERVICES in docker etcd kube-apiserver kube-controller-manager  kube-scheduler; do systemctl restart $SERVICES;systemctl enable $SERVICES;systemctl status $SERVICES; done
      ```
  - Edit the config files in /etc/kubernetes/ and start services on minions

    -
      ```
      for SERVICES in docker kube-proxy.service  kubelet.service; do systemctl restart $SERVICES; systemctl enable $SERVICES; systemctl status $SERVICES; done
      ```

# Using Kubernetes

- kubectl
  - Ask for information
    - kubectl get minions #or pods, services, replicationcontrollers
    - kubectl describe pod *name*
  - Create primitives
    - kubectl create -f yaml-or-json-information
  - Update primitives
    - kubectl update -f yaml-or-json-information

# Service example

*File httpd-service:*

```
{
  "id": "httpd",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "selector": {
    "name": "httpd"
  },
  "containerPort": 80,
  "protocol": "TCP",
  "port": 80
}
```

# services

```
# kubectl create -f httpd-service
httpd-service


# kubectl get service
NAME     LABELS    SELECTOR                     IP                  PORT
httpd              name=httpd                   10.254.0.1          80
```

# Pod example

```
{
"kind": "Pod","apiVersion": "v1beta1","id": "httpd",
    "labels": {"name": "httpd"},
    "desiredState": {"manifest": {
        "version": "v1beta1","id": "httpd","volumes": null,
        "containers": [ { "name": "master","image":"registry-host:5000/httpd:latest",
             "ports": [ { "containerPort": 80,"hostPort": 80,"protocol": "TCP" } ],
          } ],
         "restartPolicy": { "always": {} }
    }, },
}
```

# pod

```
# kubectl get pods

NAME            IMAGE(S)                     HOST                    LABELS
      STATUS

docker-registry registry:latest           rhel7-02.localdomain/ name=docker-registry
      Running

httpd        registry-host/httpd:latest rhel7-02.localdomain/   name=httpd
    Waiting

##Time Passes

# kubectl get pods

NAME            IMAGE(S)                     HOST                    LABELS
      STATUS

docker-registry registry:latest           rhel7-02.localdomain/ name=docker-registry
      Running

httpd        registry-host/httpd:latest rhel7-02.localdomain/   name=httpd
    Running
```

# Replication controller

```
{"id": "httpdController","kind": "ReplicationController","apiVersion": "v1beta1",
  "desiredState": {"replicas": 3,"replicaSelector": {"name": "httpd"},
    "podTemplate": {"desiredState": {
       "manifest": {
           "version": "v1beta1","id": "httpd",
             "containers": [{
               "name": "httpd","image": "registry-host:5000/httpd:latest",
               "ports": [{"containerPort": 80, "hostPort": 80}]
             }]
       }
    },"labels": {"name": "httpd",} }
  },
  "labels": {"name": "httpdController"}
}
```

# Replication controller

```
# kubectl get replicationcontrollers
NAME                   IMAGE(S)                    SELECTOR       REPLICAS
httpdController        rhel7.your.com/httpd:latest name=httpd     3
# kubectl get pods
NAME                                 IMAGE(S)
 HOST                    LABELS                 STATUS
docker-registry                        registry:latest
 rhel7-02.localdomain/   name=docker-registry   Running
httpd                                10.254.0.2:5000/httpd:latest
 rhel7-02.localdomain/   name=httpd             Running
7e8c6f48-9ada-11e4-b1e5-fa163ef2f051 10.254.0.2:5000/httpd:latest
 rhel7-01.localdomain/   name=httpd             Running
7e8c8d72-9ada-11e4-b1e5-fa163ef2f051 10.254.0.2:5000/httpd:latest
 <unassigned>            name=httpd             Waiting
```

# Replication controller

```
# kubectl describe replicationcontrollers httpdController
Name:        httpdController
Image(s):    10.254.0.2:5000/httpd:latest
Selector:    name=httpd
Labels:      name=httpdController
Replicas:    3 current / 3 desired
Pods Status: 2 Running / 1 Waiting / 0 Terminated
# kubectl describe pod httpd
Name:                 httpd
Image(s):             10.254.0.2:5000/httpd:latest
Host:                 rhel7-02.localdomain/
Labels:               name=httpd
Status:               Running
Replication Controllers:  httpdController (3/3 replicas created)
```

# The application is deployed

```
# curl 10.254.0.3 # <-httpd service or a specific pod-> 10.20.2.5

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">



<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">

<head>

<title>Test Page for the Apache HTTP Server on Red Hat Enterprise Linux</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<style type="text/css">
```

# Kubernetes environment variables

kubernetes and docker-links style environment variables

```
"Env": [
        "STORAGE=local",
        "STORAGE_PATH=/var/lib/docker-registry",
        "DOCKER_REGISTRY_SERVICE_HOST=10.254.0.1",
         "DOCKER_REGISTRY_SERVICE_PORT=5000",
         "DOCKER_REGISTRY_PORT=tcp://10.254.0.1:5000",
         "DOCKER_REGISTRY_PORT_5000_TCP=tcp://10.254.0.1:5000",
         "DOCKER_REGISTRY_PORT_5000_TCP_PROTO=tcp",
         "DOCKER_REGISTRY_PORT_5000_TCP_PORT=5000",
         "DOCKER_REGISTRY_PORT_5000_TCP_ADDR=10.254.0.1",
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "DOCKER_REGISTRY_CONFIG=/docker-registry/config/config_sample.yml",
        "SETTINGS_FLAVOR=dev"
     ],
```

# Use Kubernetes environment variables

Use the kubernetes and docker-links style environment variables as part of the startup script or commands for multi-tier applications.

- Build up the config file in a startup script for the platform in the container
- Directly call environment variables in the startup command

# Kubernetes network concepts

- Every pod is given an ip address to avoid collisions in ports
- Every pod can talk to every other pod in the container network
- Service ips are used to make more persistent connections as pods come and go
- Service ips are iptables rules to route traffic on hosts running kube-proxy and on the layer 2 container network
- External ip addresses can also be specified for services so that external hosts to the container network can route to the containers.  These are where port collisions can still occur on the “public” ip of the container host.

# Kubernetes network topology

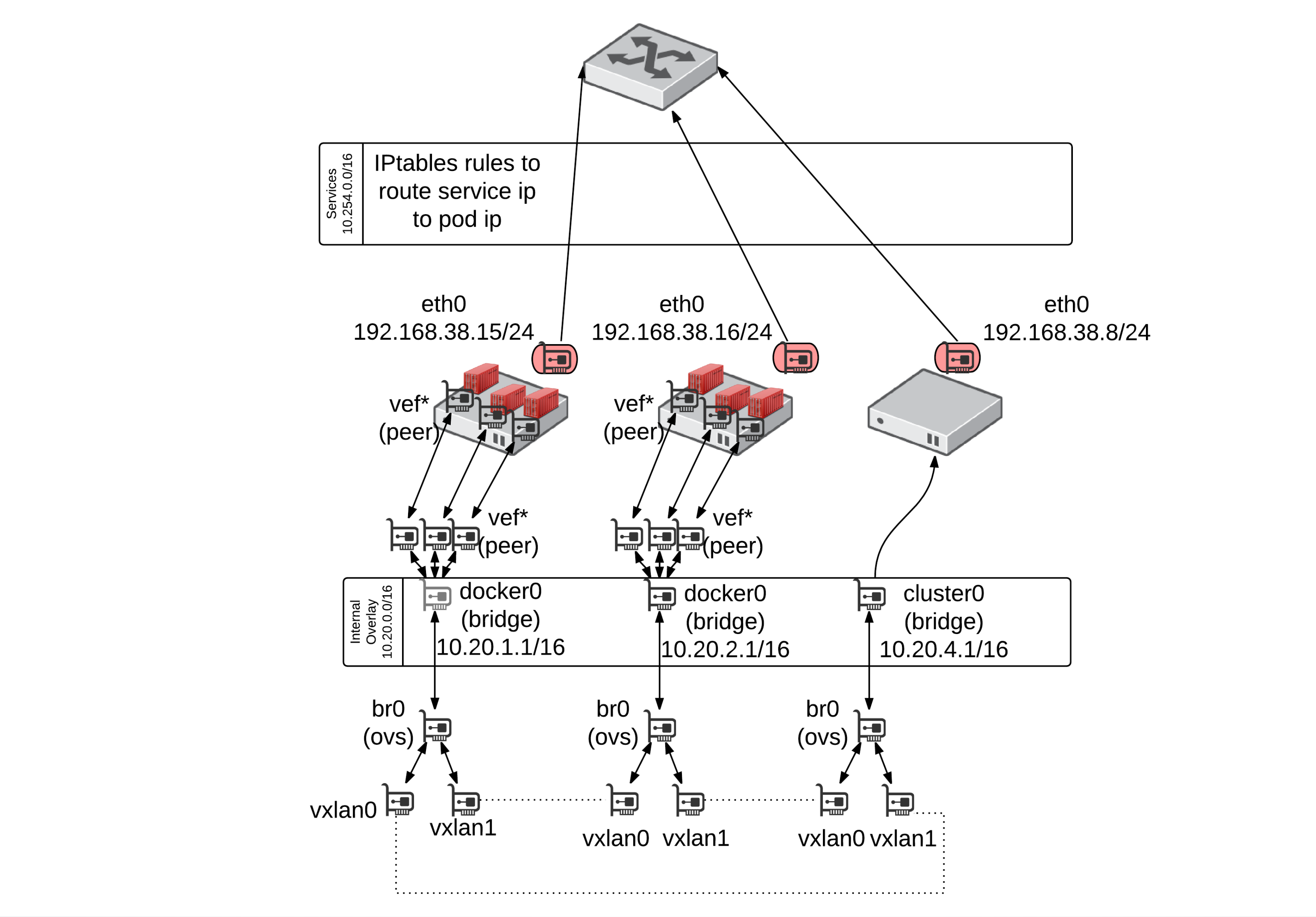
**For my lab example:**
Using a vxlan tunnel overlay network for
minions and controller
3 normal network devices + 12 virtual network devices
+ 2 x running containers network devices

Running as openstack instances
Using openstack neutron for private network
and public ip routing to instances

NET-CEPTION:

containers connect to peer interfaces connected...
to peer interfaces on the docker bridge ...
connected to an openvswitch ...
with a mesh of vxlan tunnels to the private ips

# Kubernetes network topology



# Kubernetes network example

10.20.0.0/16 is allocated for the container network

Create linux bridge docker0 manually and assign 10.20.**1**.1/16 on the first host

```
brctl addbr docker0 #create a bridge with a real “fake” mac
brctl stp docker0 on #turn on spanning tree to prevent loops
ip addr add 10.20.1.1/16 dev docker0 #give the bridge an ip address
ifconfig docker0 mtu public-50 #adjust mtu for the tunnel overhead of vxlan
```

Docker takes a /24 within the /16 and uses the existing docker0 bridge:

Add to /etc/sysconfig/docker OPTIONS line:

```
  --fixed-cidr=10.20.1.1/24 --bridge=docker0
```

# Kubernetes network example

Create an openvswitch vxlan mesh network via tunnels between hosts via the openstack private ip

```
ovs-vsctl add-br br0

ovs-vsctl add-port br0 gre0 -- set interface vxlan0 type=vxlan options:remote_ip=192.168.38.8

ovs-vsctl add-port br0 gre0 -- set interface vxlan1 type=vxlan options:remote_ip=192.168.38.15
```

Add openvswitch to the docker0 bridge

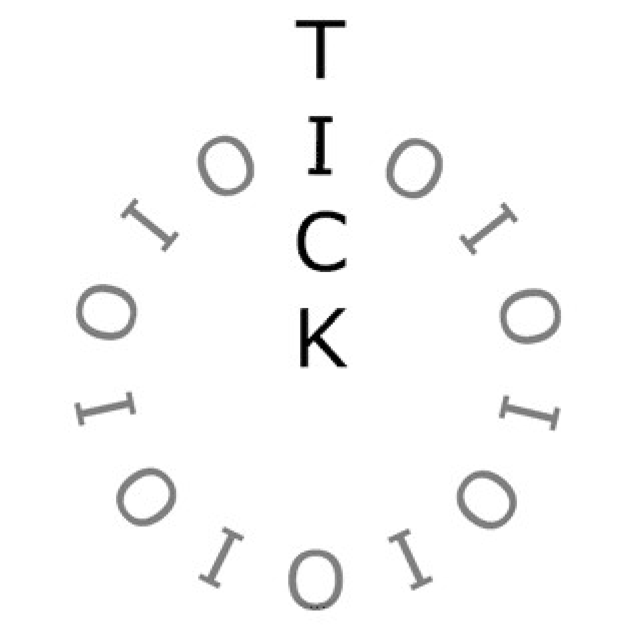
```
brctl addif docker0 br0
```

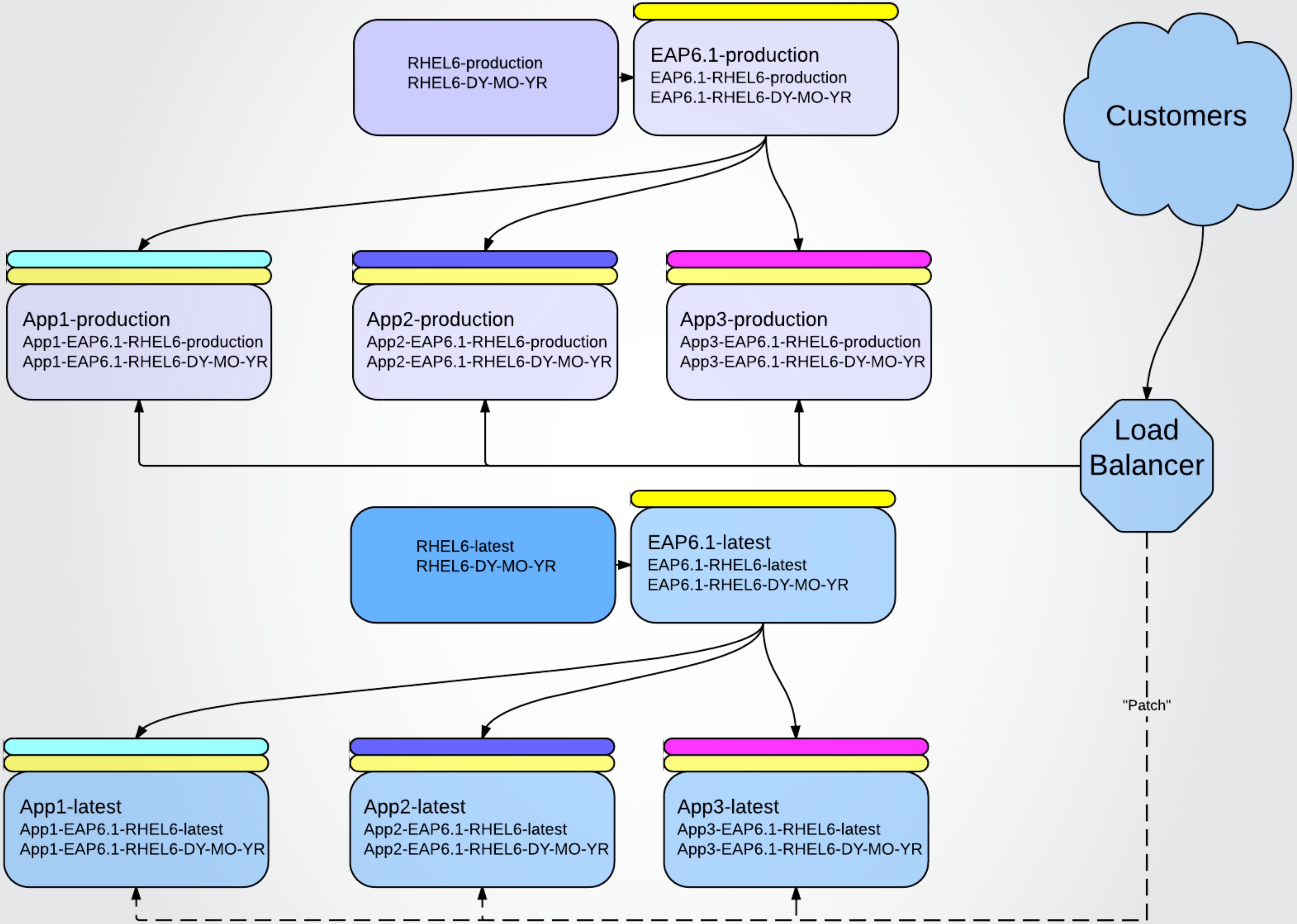
Bring it up:

```
ifconfig br0 mtu public-50 #adjust mtu for the tunnel overhead of vxlan

ifconfig br0 up #be patient for stp to complete in a few seconds
```

# Patch cycle becomes a cutover (Docker)





RHEL6-production
RHEL6-DY-MO-YR

EAP6.1-production
EAP6.1-RHEL6-production
EAP6.1-RHEL6-DY-MO-YR

Customers

App1-production
App1-EAP6.1-RHEL6-production
App1-EAP6.1-RHEL6-DY-MO-YR

App2-production
App2-EAP6.1-RHEL6-production
App2-EAP6.1-RHEL6-DY-MO-YR

App3-production
App3-EAP6.1-RHEL6-production
App3-EAP6.1-RHEL6-DY-MO-YR

Load Balancer

RHEL6-latest
RHEL6-DY-MO-YR

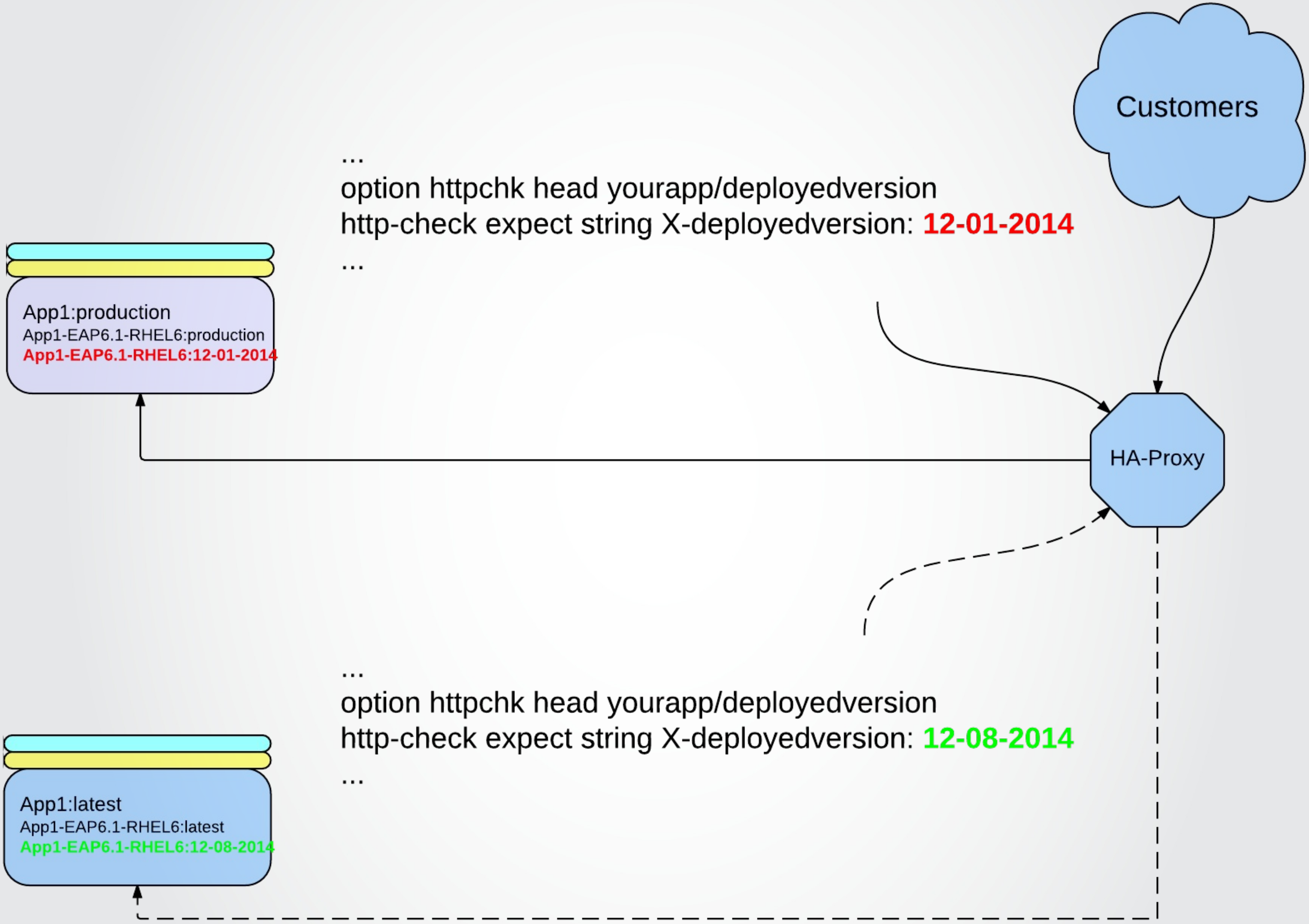
EAP6.1-latest
EAP6.1-RHEL6-latest
EAP6.1-RHEL6-DY-MO-YR

"Patch"

App1-latest
App1-EAP6.1-RHEL6-latest
App1-EAP6.1-RHEL6-DY-MO-YR

App2-latest
App2-EAP6.1-RHEL6-latest
App2-EAP6.1-RHEL6-DY-MO-YR

App3-latest
App3-EAP6.1-RHEL6-latest
App3-EAP6.1-RHEL6-DY-MO-YR



Customers

```
...
option httpchk head yourapp/deployedversion
http-check expect string X-deployedversion: 12-01-2014
...
```

App1:production
App1-EAP6.1-RHEL6:production
**App1-EAP6.1-RHEL6:12-01-2014**

HA-Proxy

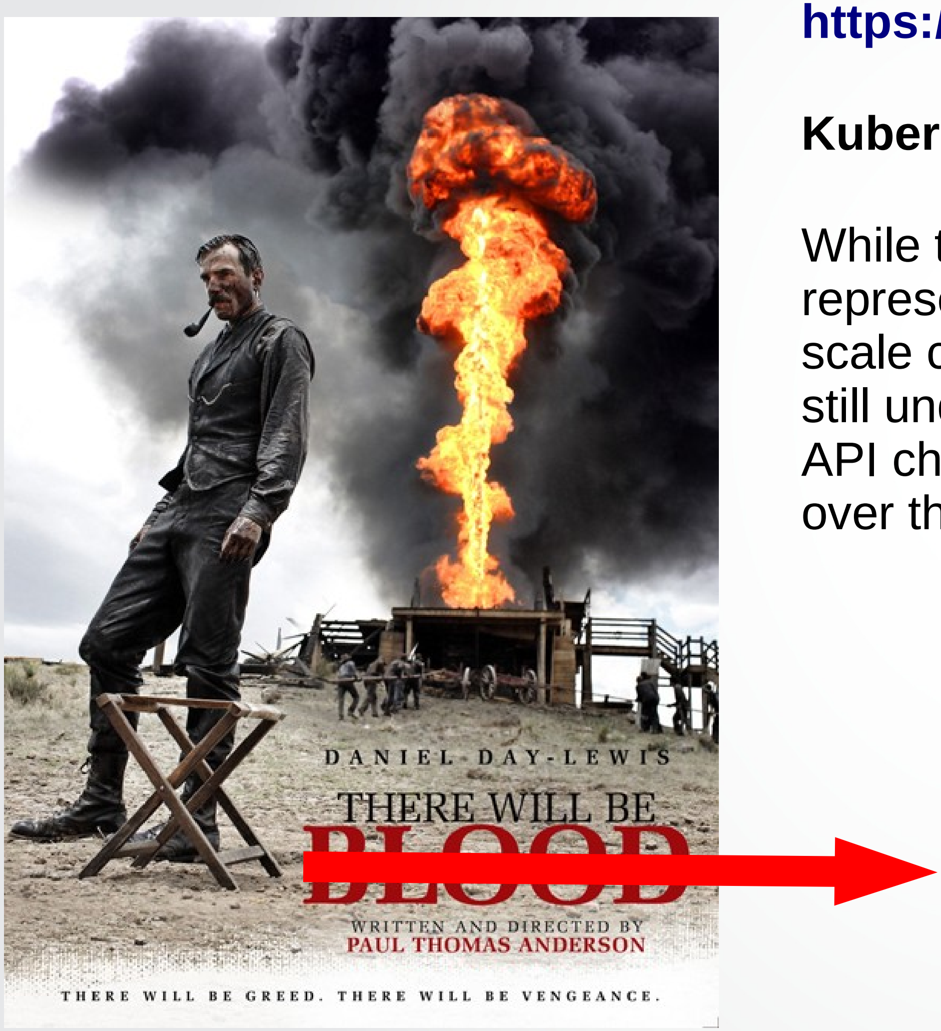
```
...
option httpchk head yourapp/deployedversion
http-check expect string X-deployedversion: 12-08-2014
...
```

App1:latest
App1-EAP6.1-RHEL6:latest
**App1-EAP6.1-RHEL6:12-08-2014**

# Kubernetes patch cycle

- Replication controller rolling updates
  - Replication controller for production – N copies
- Rolling upgrade starts – both replication controllers are selected by the same service
  - Replication controller for production – N – 1 copies
  - Replication controller for next version of production – 1 copy
- ...Repeat until...
- Upgrade finishes
  - Replication controller for production (old) – deleted after 0 copies
  - Replication controller for current version in prodution – N copies

# Is Kubernetes production ready?



**Bugs**

**https://github.com/GoogleCloudPlatform/kubernetes**

**Kubernetes is in pre-production beta!**

While the concepts and architecture in Kubernetes represent years of experience designing and building large scale cluster manager at Google, the Kubernetes project is still under heavy development. Expect bugs, design and API changes as we bring it to a stable, production product over the coming year.

# What’s next?

- Scheduler - Apache Mesos
  - Mesos will handle YARN and Kubernetes jobs to best utilize resources
  - You can try it out, as a docker image of course
- https://github.com/mesosphere/kubernetes-mesos

# What’s next?

- Greater networking agility
  - Flannel (formerly rudder)
  - Dynamic assignment of /24 networks
  - UDP based overlay network automation
  - https://github.com/coreos/flannel

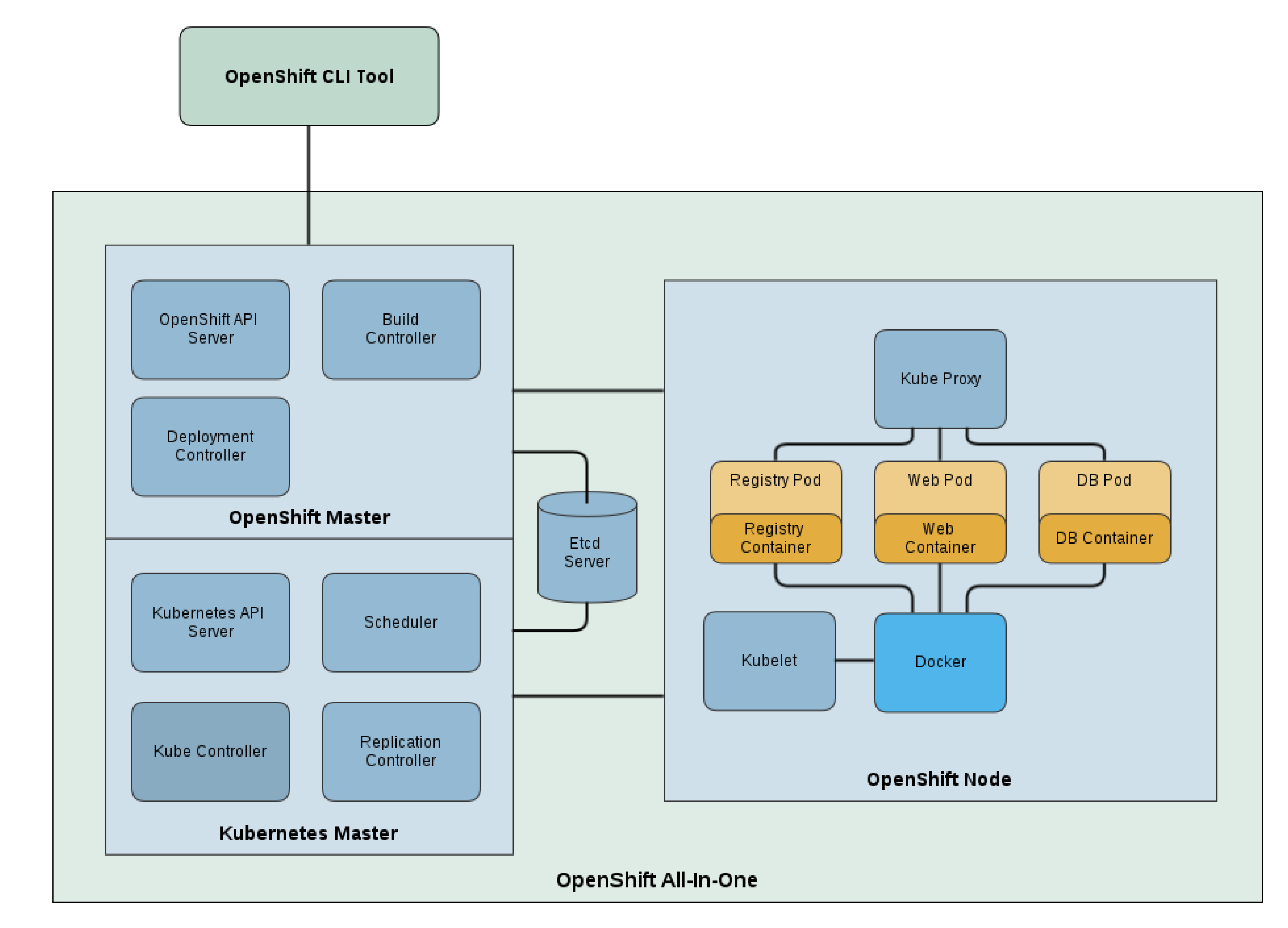
# RHEL Atomic Beta

- Container Oriented OS based on RHEL7 and Project Atomic
  - Docker and Kubernetes
  - All applications are to run in a container
- Atomic updates of the OS
  - rpm-ostree

# OpenShift 3.0 roadmap

- RHEL atomic are the OpenShift nodes
  - Overlay network technology TBD
- Image builds via kubernetes
  - https://blog.openshift.com/openshift-v3-deep-dive-docker-kubernetes/
- Source-to-Image builds via kubernetes
  - https://blog.openshift.com/builds-deployments-services-v3/

# OpenShift 3.0 architecture



# OpenShift 3.0 build

Builds a docker image via a dockerfile in a github repo

```
{
“id”: “build100″,“kind”: “BuildConfig”,“apiVersion”: “v1beta1″,
  “desiredInput”: {
    “type”:“docker”,“sourceURI”: “git://github.com/bparees/openshift3-blog-part1.git”,
    “imageTag”:  “openshift/origin-ruby-sample”,“registry”:  “127.0.0.1:5001″
  },
  “secret”: “secret101″
}
```

# OpenShift 3.0 sti

```
{
“id”: “ruby-sample-build”,“kind”: “BuildConfig”,“apiVersion”: “v1beta1″,
  “parameters”: {
    “source” : {“type” : “Git”,“git” : {“uri”: “git://github.com/openshift/ruby-hello-world.git”} },
    “strategy”: {“type”: “STI”,“stiStrategy”: {“builderImage”: “openshift/ruby-20-centos”} },
    “output”: { “imageTag”: “openshift/origin-ruby-sample:latest”,“registry”: “172.121.17.1:5001″},
   },
  “secret”: “secret101″,
  “labels”: {“name”: “ruby-sample-build”}
}
```

redhat®