

RED HAT
SUMMIT

BOSTON, MA
JUNE 23-26, 2015

Jenkins for continuous delivery of infrastructure via Docker

Greg Hoelzer and Michael Heldebrant
Red Hat
June 2015

AGENDA

- CI/CD for Infrastructure
- Docker Tagging and Genealogy
- Testing in the context that is deployed to production
- Docker factory floor from OS image to production
 - Build Slaves
 - Dockerfile builds via Jenkins
 - Docker container build slaves for Jenkins
 - Production image build
- Satellite 6 and container builds
- Patch cycle with Containers

An allegory for managing constant change in operations



Traditional patch cycle in a downtime window



Continuous Delivery

CI/CD for infrastructure?

Challenge: How can you continuously deliver and integrate the latest infrastructure and platforms to deploy and maintain your applications?

Solution: By continuously updating and deploying infrastructure to build and test your applications that you can then deploy in a single unit to production.

Docker containers are one way to approach this solution.

Tagging



Docker tags

By using symbolic tags you automatically update your inputs to the next step of the factory build with Dockerfiles via the FROM value

For example:

OS:latest – always the most current OS update that finished a docker build

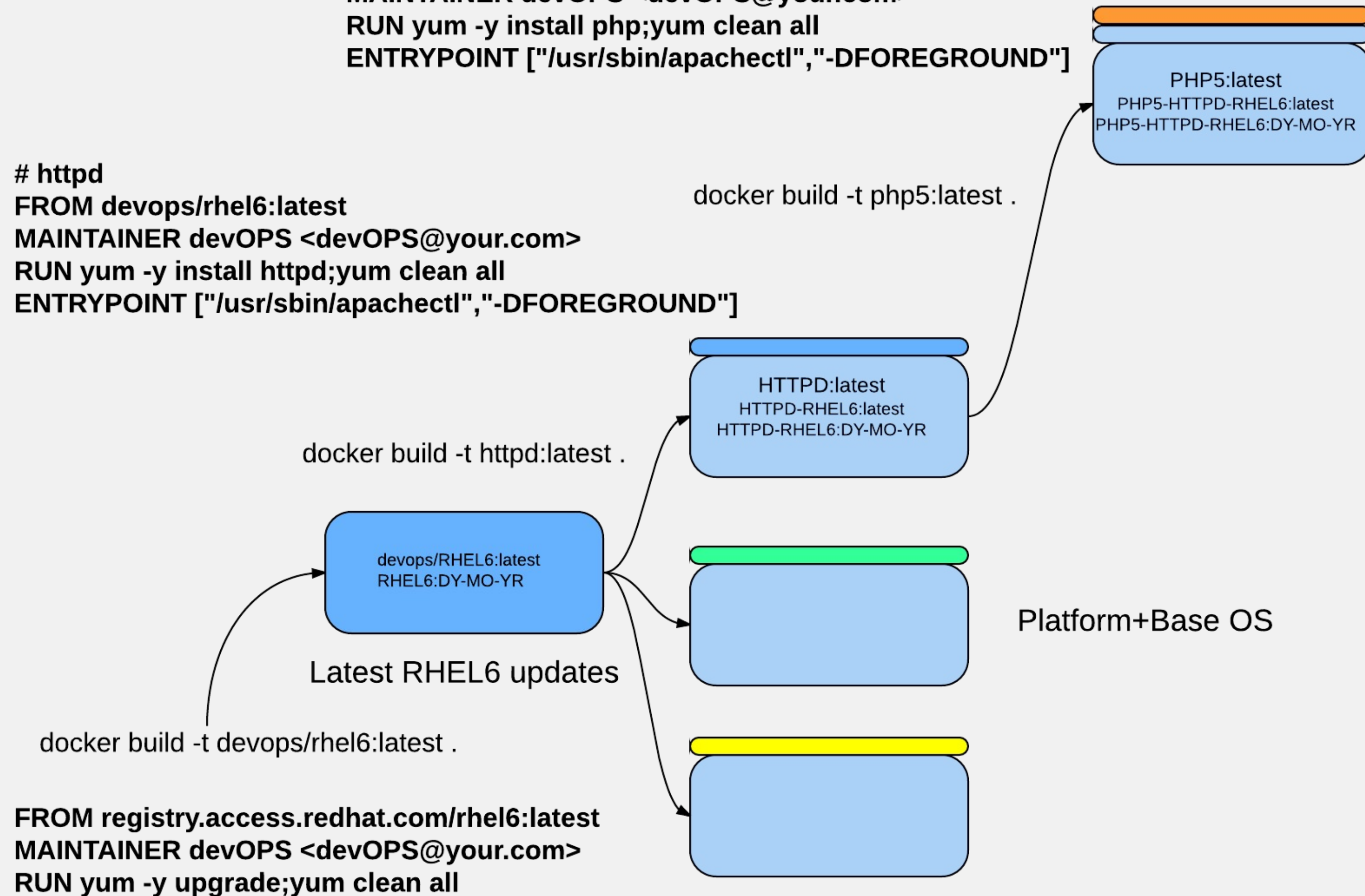
Platform:latest – the most current platform installed onto OS:latest

YourApp:latest – the most current application deployed on Platform:latest

Inputs update automatically via tags

```
# php5 + httpd  
FROM httpd:latest  
MAINTAINER devOPS <devOPS@your.com>  
RUN yum -y install php;yum clean all  
ENTRYPOINT ["/usr/sbin/apachectl","-DFOREGROUND"]
```

```
# httpd  
FROM devops/rhel6:latest  
MAINTAINER devOPS <devOPS@your.com>  
RUN yum -y install httpd;yum clean all  
ENTRYPOINT ["/usr/sbin/apachectl","-DFOREGROUND"]
```



```
FROM registry.access.redhat.com/rhel6:latest  
MAINTAINER devOPS <devOPS@your.com>  
RUN yum -y upgrade;yum clean all
```


Docker tags:versions

Multiple factory lines can coexist. All Docker images that are the same input and steps are cached and reused in the build process.

For example:

YourApp:Latest – always the most current update that finished a Docker build

YourApp:QA – the version undergoing testing

YourApp:Tested – the version that passed integrated testing

YourApp:Production – the most current version running in production

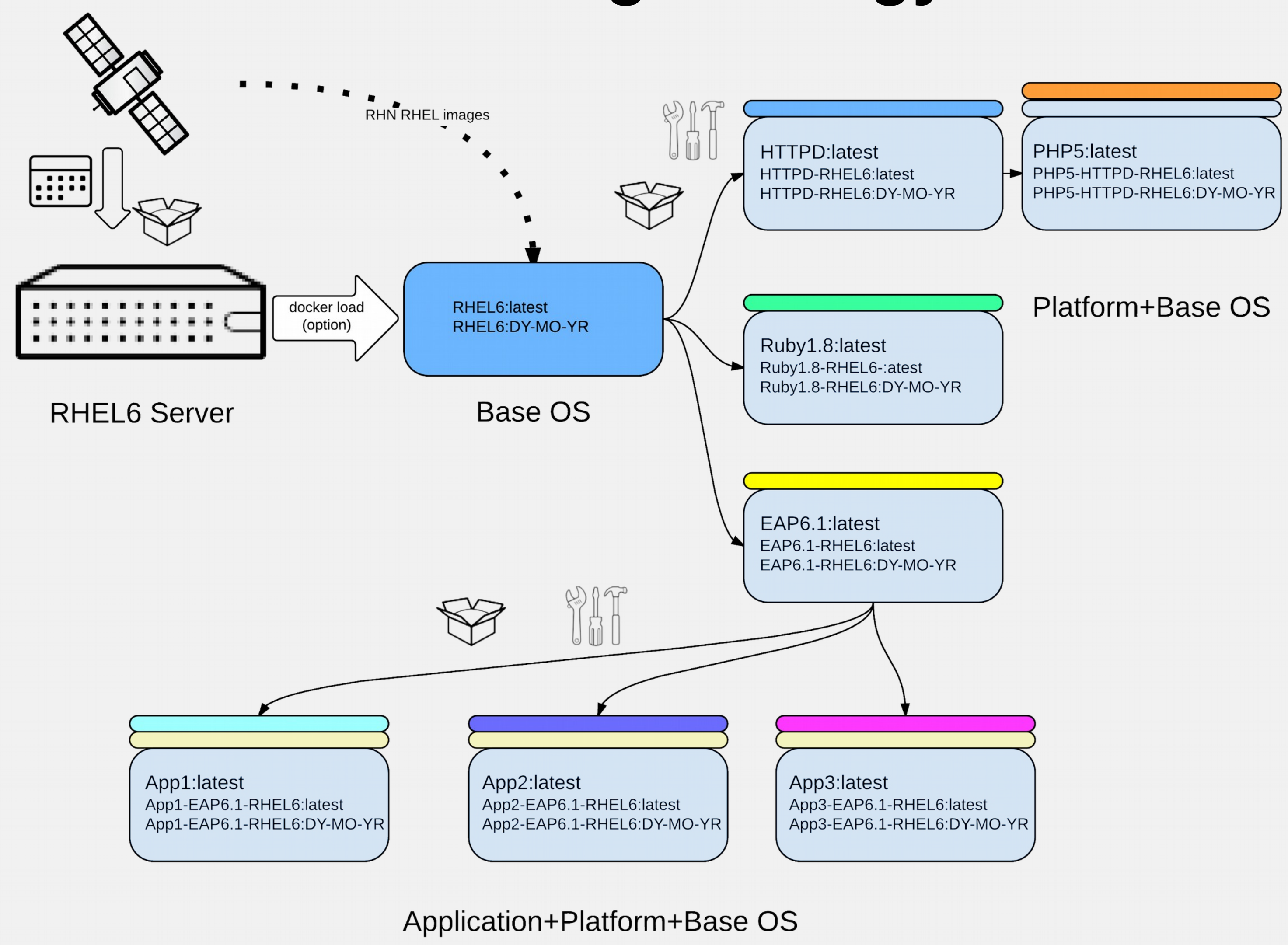
YourApp:Next – the next version to deploy to production

YourApp:Tested-datestamp – save a tag by date that passed the build and testing for archive

Genealogy



Docker genealogy



Testing



Has this ever happened to you?

Don't act like a bond villan DevOps team, take full advantage of your new capabilities:

Dr. Evil: All right guard, begin the unnecessarily slow-moving dipping mechanism.
[guard starts dipping mechanism]

Dr. Evil: Close the tank!

Scott Evil: Wait, aren't you even going to watch them? They could get away!

Dr. Evil: No no no, I'm going to leave them alone and not actually witness them dying, I'm just gonna assume it all went to plan.

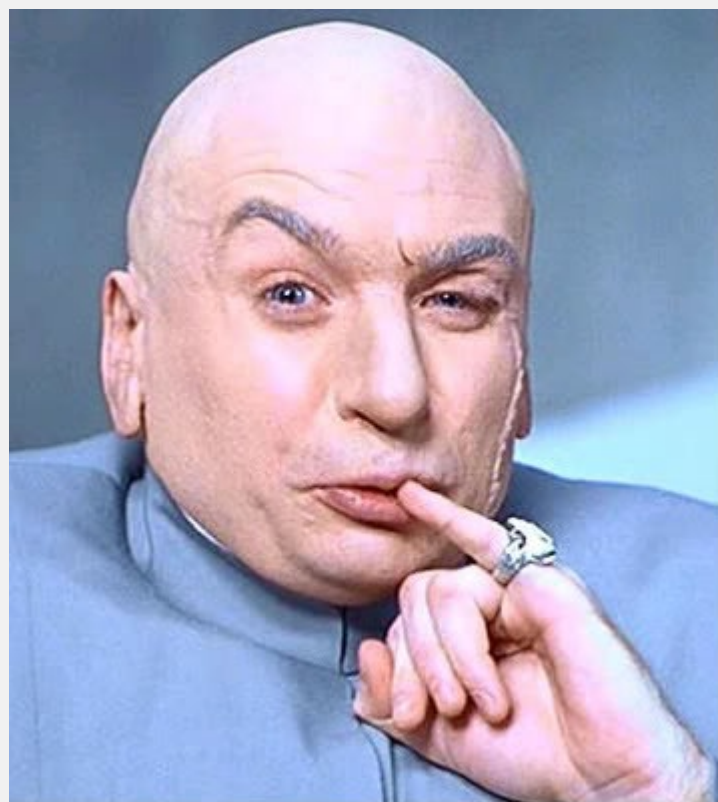


The Old way:

Of course the app was “tested” before the production deployment! It worked on my laptop!

Test at multiple levels

- Unit test your applications during the build in the same context as production will use
- Plan for automated integration tests: launch a whole app stack of containers, even database containers, and a driver container to run the tests
- Reduce human interventions as much as possible as it will become a bottleneck



New way:

Developers are responsible for writing tests to catch broken apps before deployment

Operations provides correct testing OS+Platform containers

Factory



Build a Docker factory

How does CI and Containers work together?

Define your jobs in Jenkins to build the next dependent job

- Jenkins will build from the point of change all the way to the end of the factory
- Fan out strategy for variants you need to support such as multiple jdk versions
- Fan out for for multiple os versions or patch levels, latest/tested/production/next











Docker run build and test containers via symbolic tags

- Restart containers with new code early and often in the latest track
- Restart containers with the latest versions of other tracks as they are promoted
- Always make sure you can build an app only change in the production and next tracks

Requirements to be a build slave

- Remote access
 - ssh access is easy to set up
 - configure credentials in jenkins
 - password
 - ssh keys
- Be able to run slave.jar
 - CPU
 - RAM
 - Java JRE
- Be able to build your projects
 - Maven
 - Git
 - GCC
 - etc

Containers can be build slaves:

S	Name ↓	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	master	Linux (amd64)	In sync	4.99 GB	0 B	4.99 GB	0ms 
	mvnbl-dk1.6-rhel6	Linux (amd64)	In sync	8.49 GB	0 B	8.49 GB	23ms 
	mvnbl-dk1.6-rhel7	Linux (amd64)	In sync	8.50 GB	0 B	8.50 GB	22ms 
	mvnbl-dk1.7-rhel6	Linux (amd64)	In sync	8.34 GB	0 B	8.34 GB	22ms 
	mvnbl-dk1.7-rhel7	Linux (amd64)	In sync	8.59 GB	0 B	8.59 GB	22ms 
Data obtained		4 min 11 sec	4 min 11 sec	4 min 11 sec	4 min 11 sec	4 min 11 sec	4 min 11 sec

[Refresh status](#)

Test your builds on the same versions as deployments

Project name

Description

[Escaped HTML] [Preview](#)

Discard Old Builds [?](#)

GitHub project

This build is parameterized [?](#)

Permission to Copy Artifact [?](#)

Disable Build (No new builds will be executed until the project is re-enabled.) [?](#)

Execute concurrent builds if necessary [?](#)

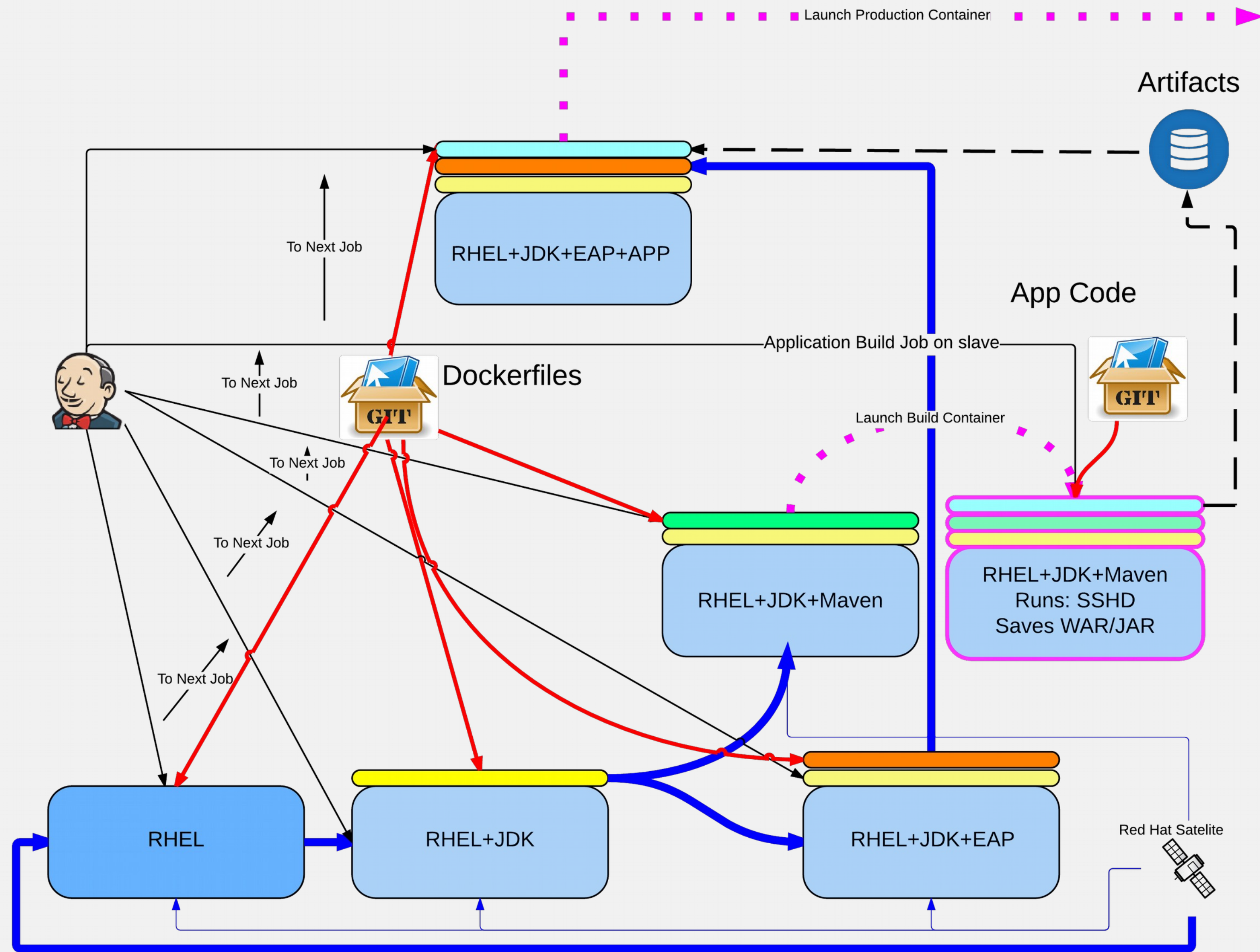
Restrict where this project can be run [?](#)

Label Expression

Slaves in [label](#): 1

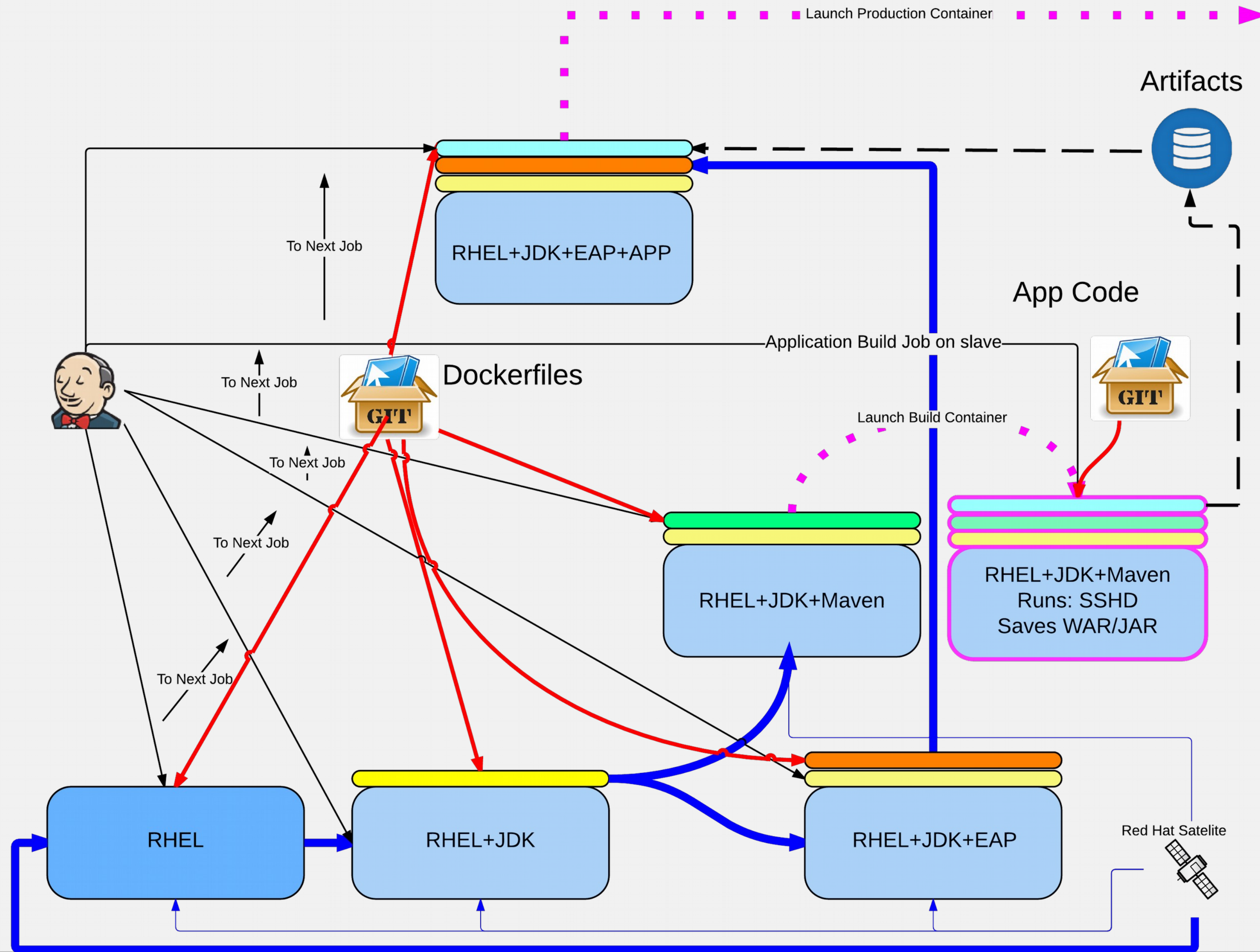
- Create build slaves that share the common ancestor image of the deployment container
- Unit tests should be written that catch changes in operating system or platform that break the application
- Restrict build jobs to the proper labeled build slave

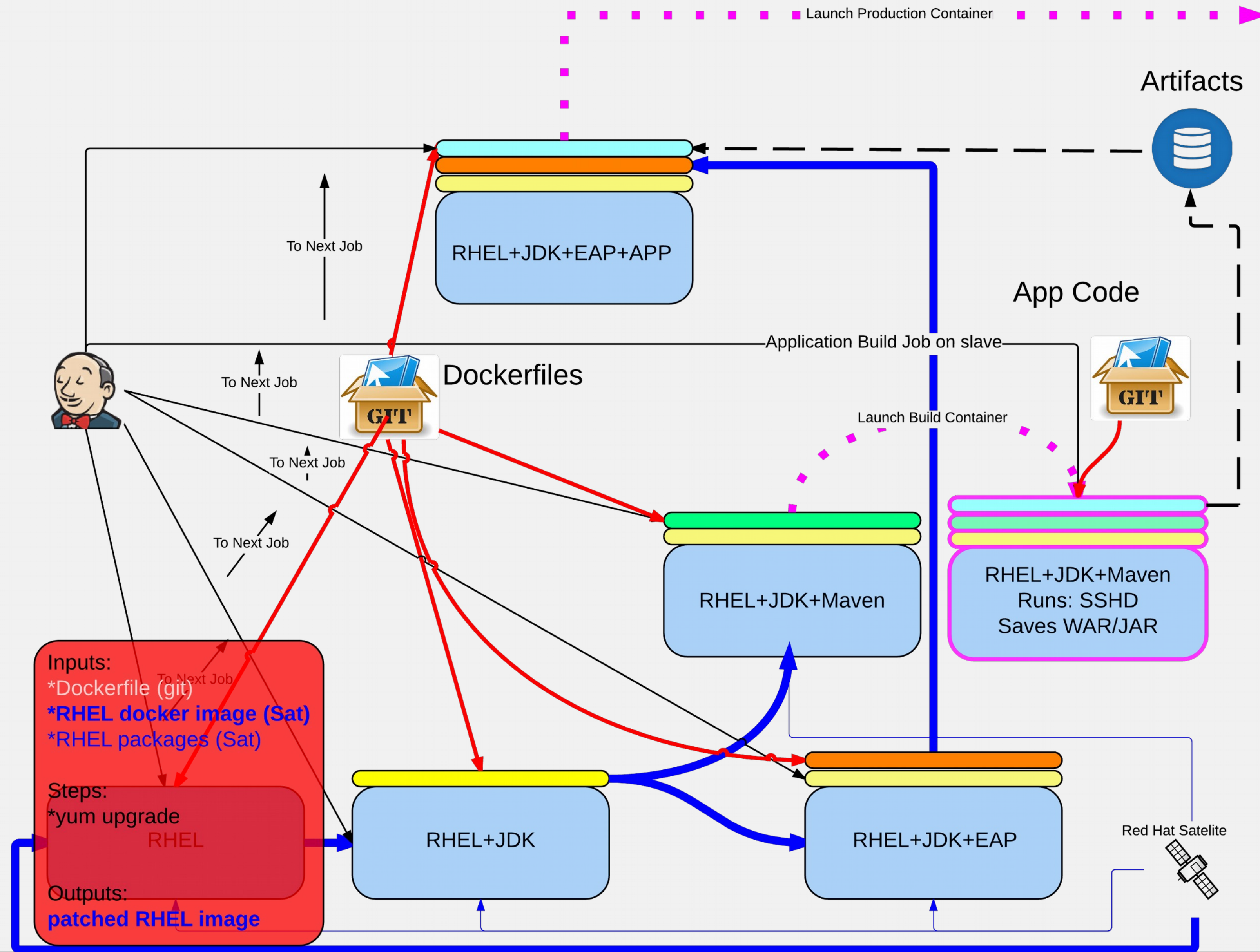
Factory Overview

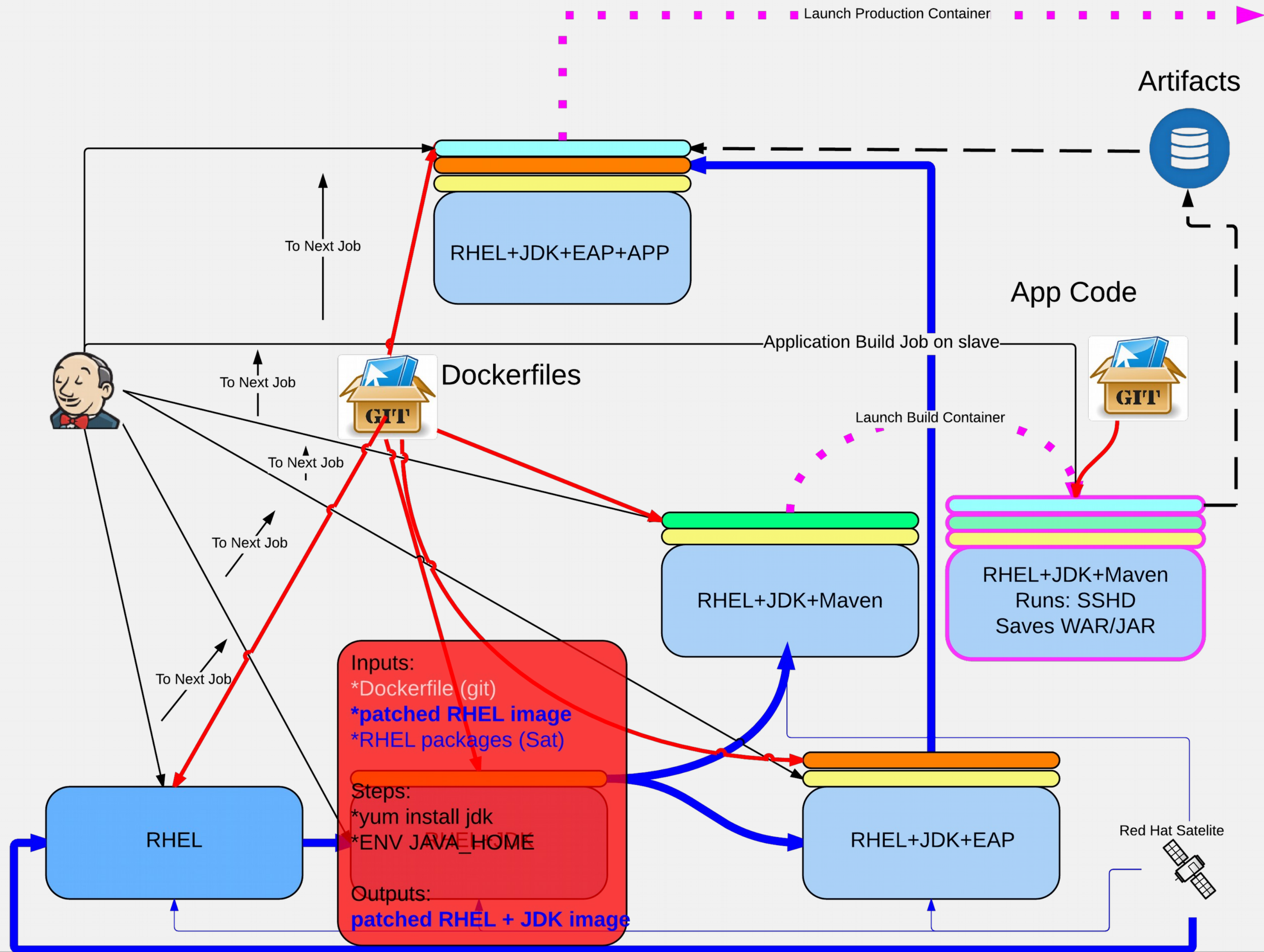


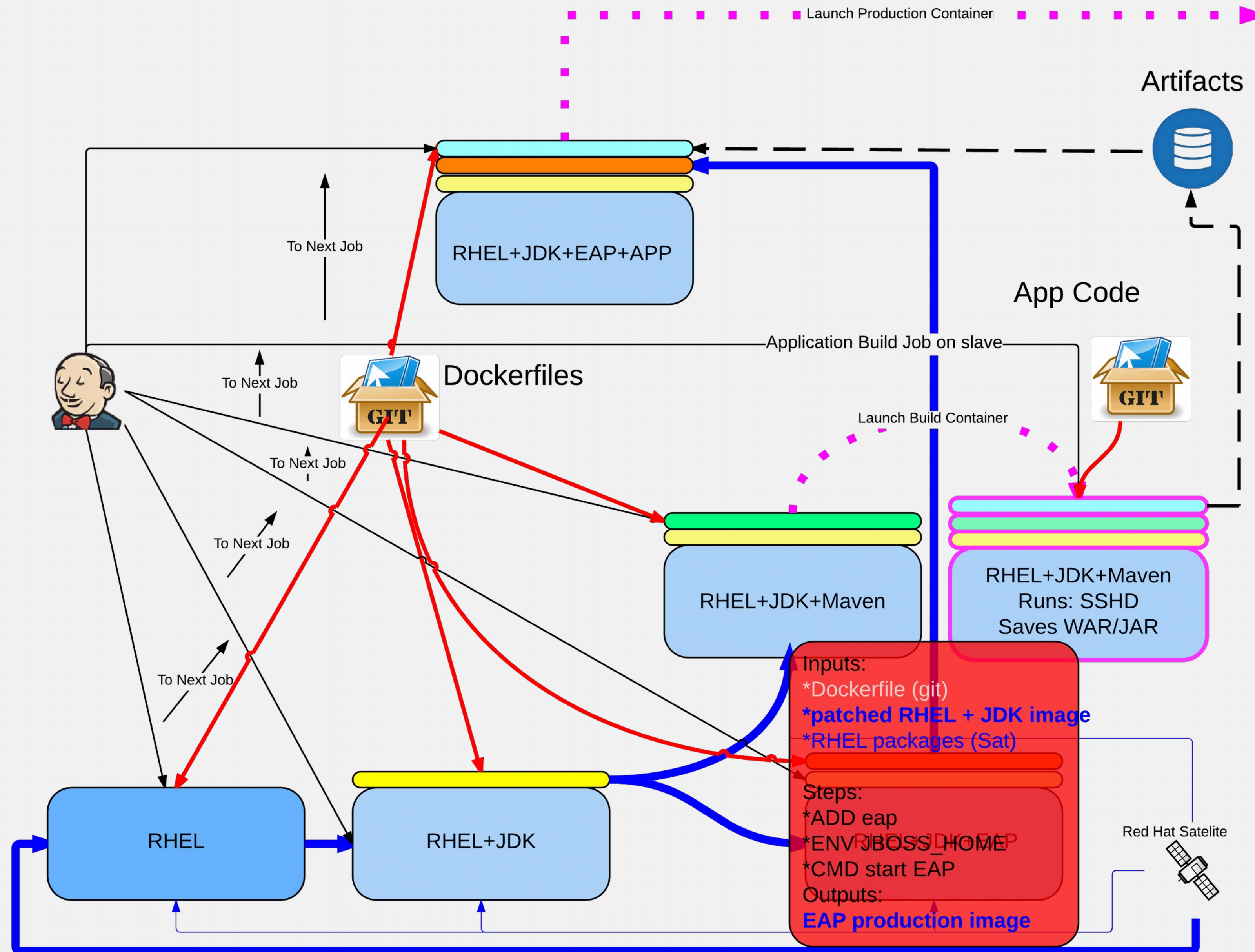
Don't Panic: Step by Step walk through

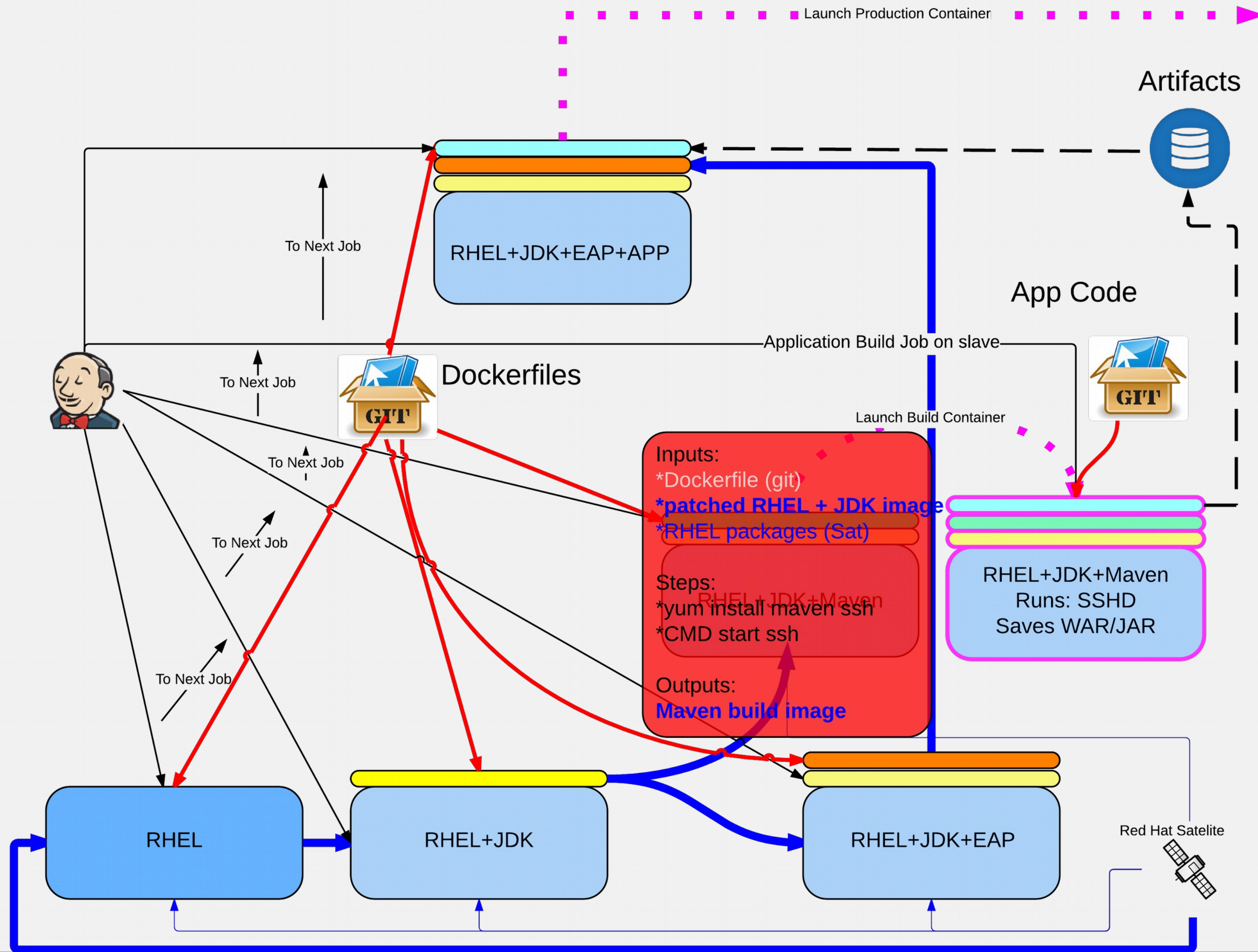


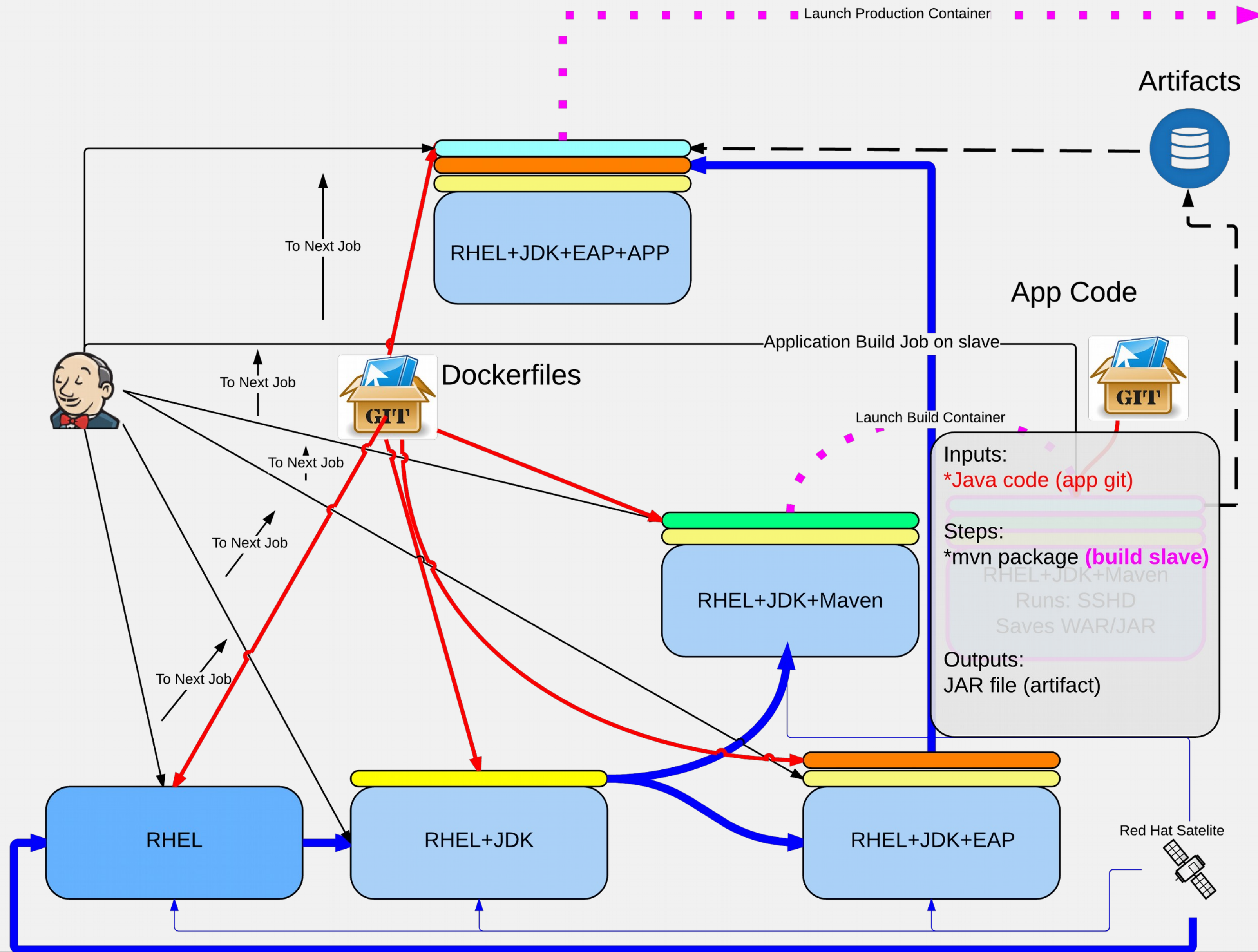


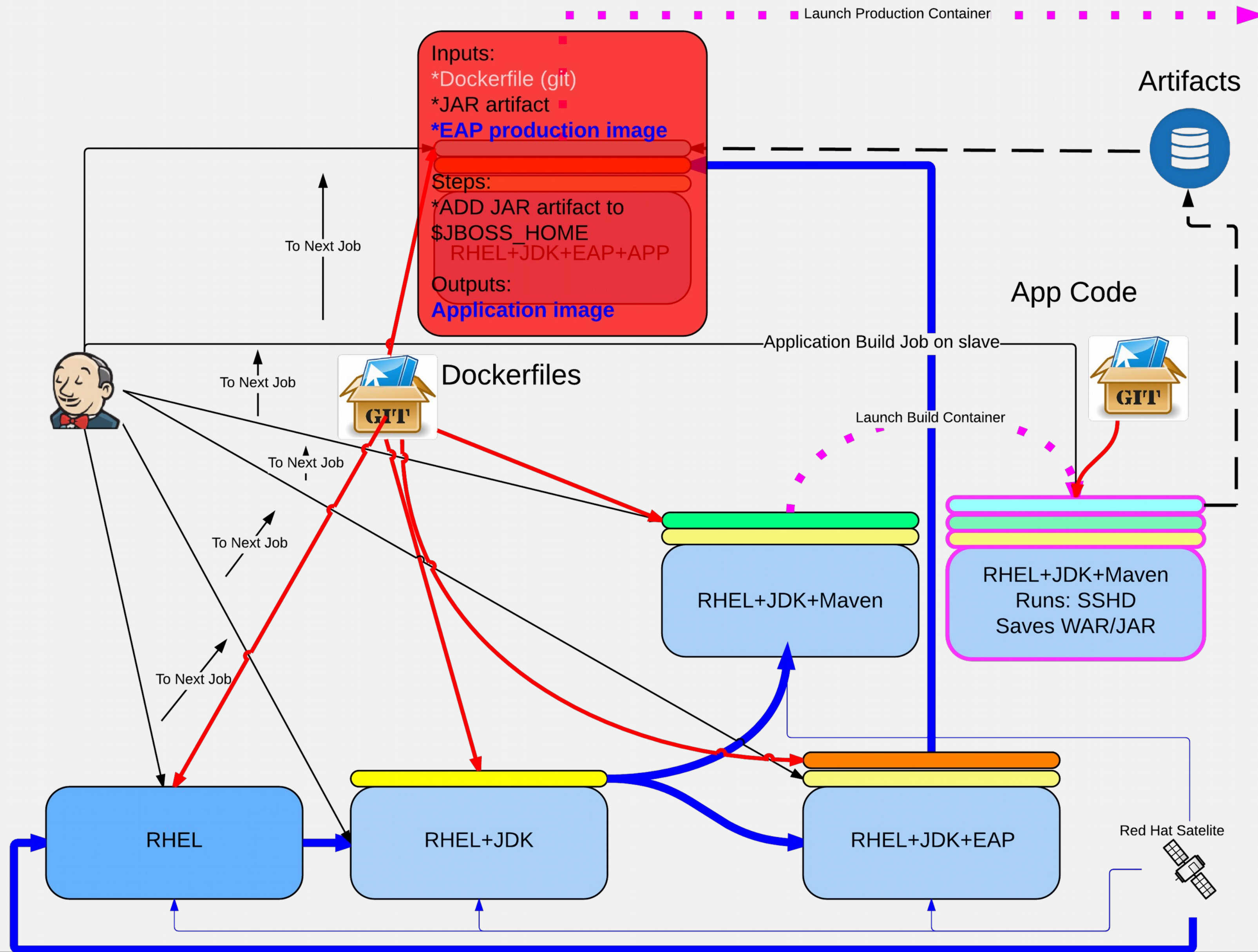














-  [Back to Project](#)
-  [Status](#)
-  [Changes](#)
-  [Console Output](#)
-  [Edit Build Information](#)
-  [Delete Build](#)
-  [Git Build Data](#)
-  [No Tags](#)
-  [See Fingerprints](#)
-  [Previous Build](#)

 **Build #241 (Mar 11, 2015 3:04:12 PM)**

No changes. Changes in dependency

1. [bld-helloworld-jdk1.6-rhel6 #102](#) → [#103](#) ([detail](#))



Started by upstream project [bld-helloworld-jdk1.6-rhel6](#) build number [103](#)
originally caused by:

- Started by upstream project [mvnbld-jdk1.6-rhel6-start](#) build number [118](#)
originally caused by:
 - Started by upstream project [mvnbld-jdk1.6-rhel6-stop](#) build number [108](#)
originally caused by:
 - Started by upstream project [mvnbld-jdk1.6-rhel6](#) build number [110](#)
originally caused by:
 - Started by upstream project [jdk1.6-rhel6](#) build number [110](#)
originally caused by:
 - Started by upstream project [rhel6-latest](#) build number [106](#)
originally caused by:
 - Started by timer



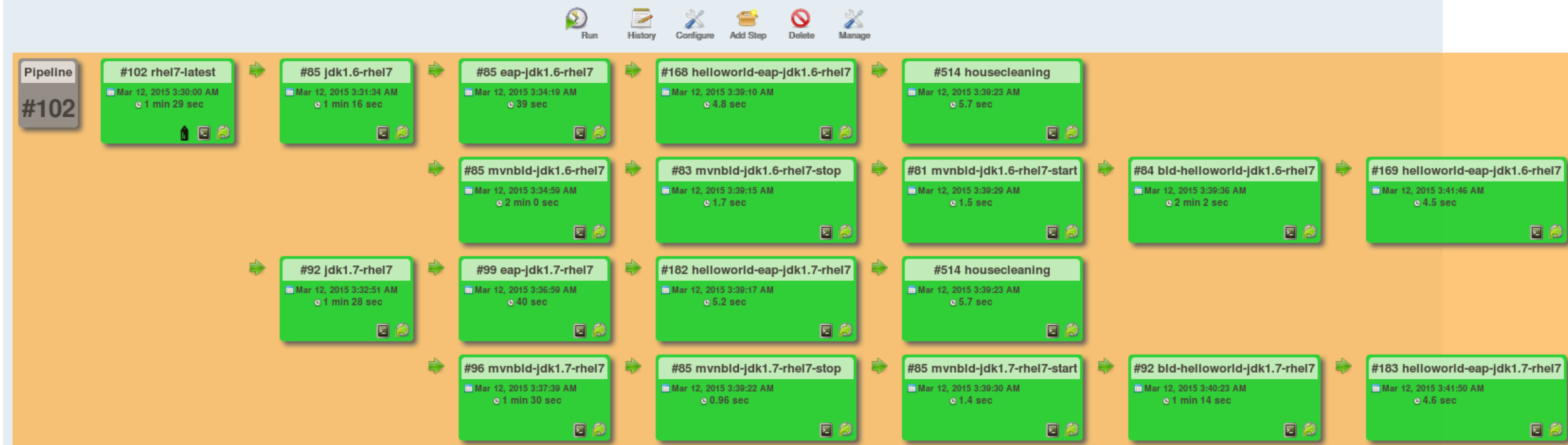
Revision: a43a468e9362209d3296943f383b7c86abb9b643

- [refs/remotes/origin/rhel6](#)

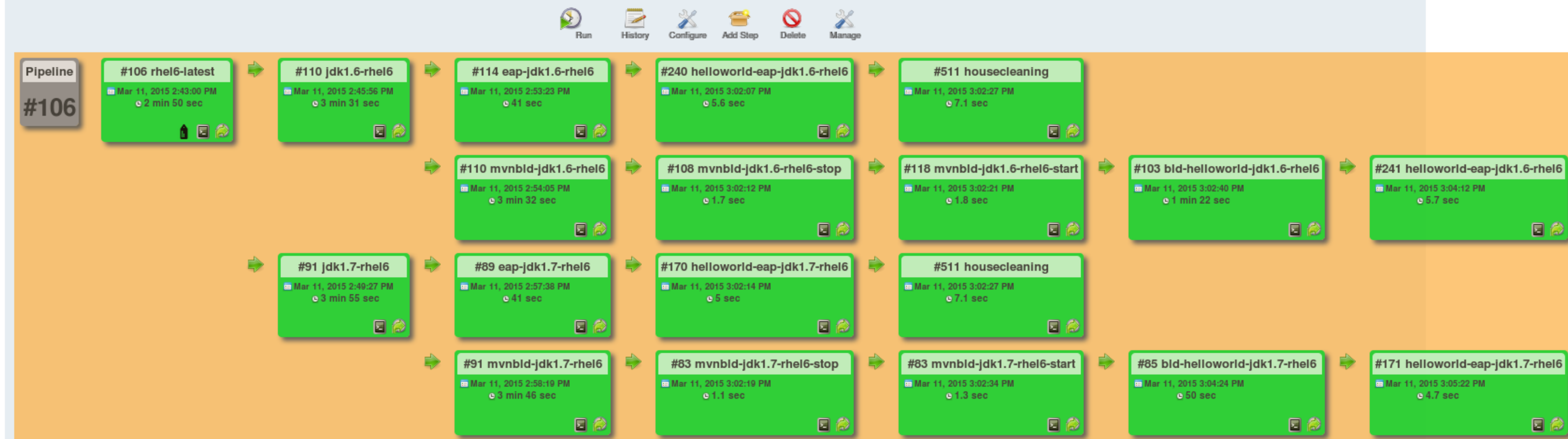
Upstream Builds

[bld-helloworld-jdk1.6-rhel6](#) [#103](#)

Build Pipeline



Build Pipeline



Satellite 6 and containers

Subscription Management for containers

Using the RHEL 7 subscription model, to create Docker images or containers, you must properly subscribe the container server on which you build them.

If you use the Red Hat registry.access.redhat.com docker images, when you use yum within the container to add or upgrade packages, the container automatically has access to the repositories available to the RHEL 7 host.

The containers can get RPM packages from the appropriate repositories so that RHEL6 image and RHEL7 image containers can co-exist on the same RHEL7 container host.

Satellite 6 for containers

In Satellite 6 create a composite content view that includes your:

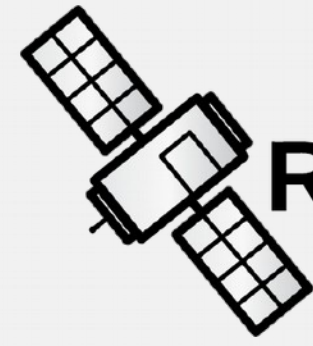
- RHEL6 content view
- RHEL7 content view

Create activation keys

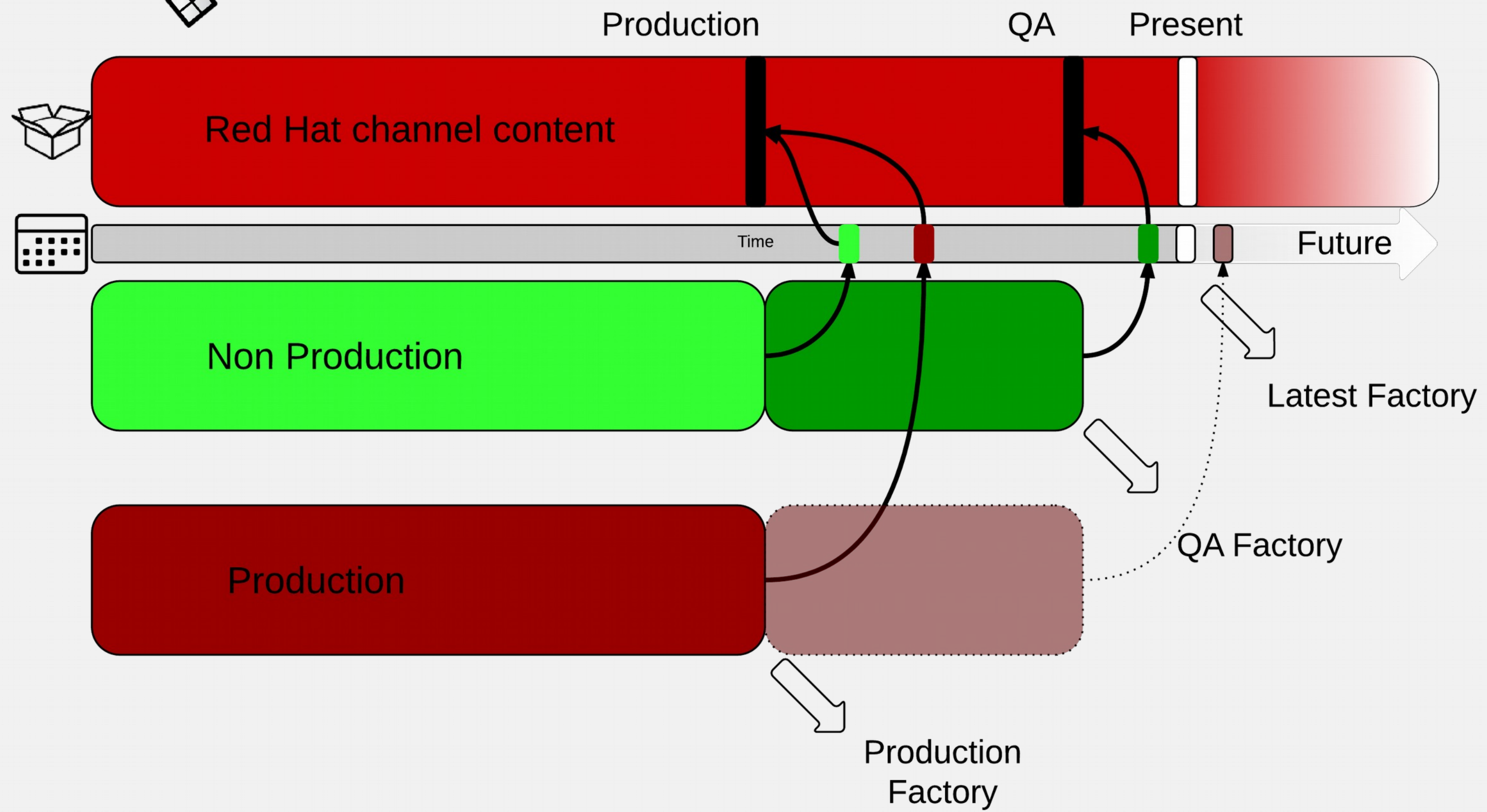
- Selects the composite content view and proper lifecycle environment
- Only the RHEL7 repos are set to enabled for the host
- Use for a RHEL7 container host during provisioning

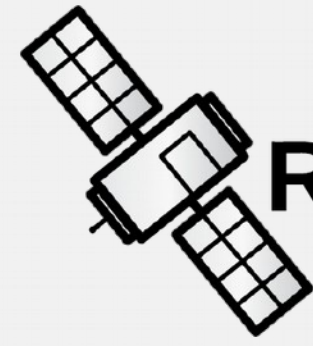
Containers can use RHEL6 and RHEL7 repos on this host with the advantage of content management capabilities

- Library matches latest content from Red Hat Network
- Lifecycle environments can match your needs for managed changes
 - QA
 - Next
 - PRD

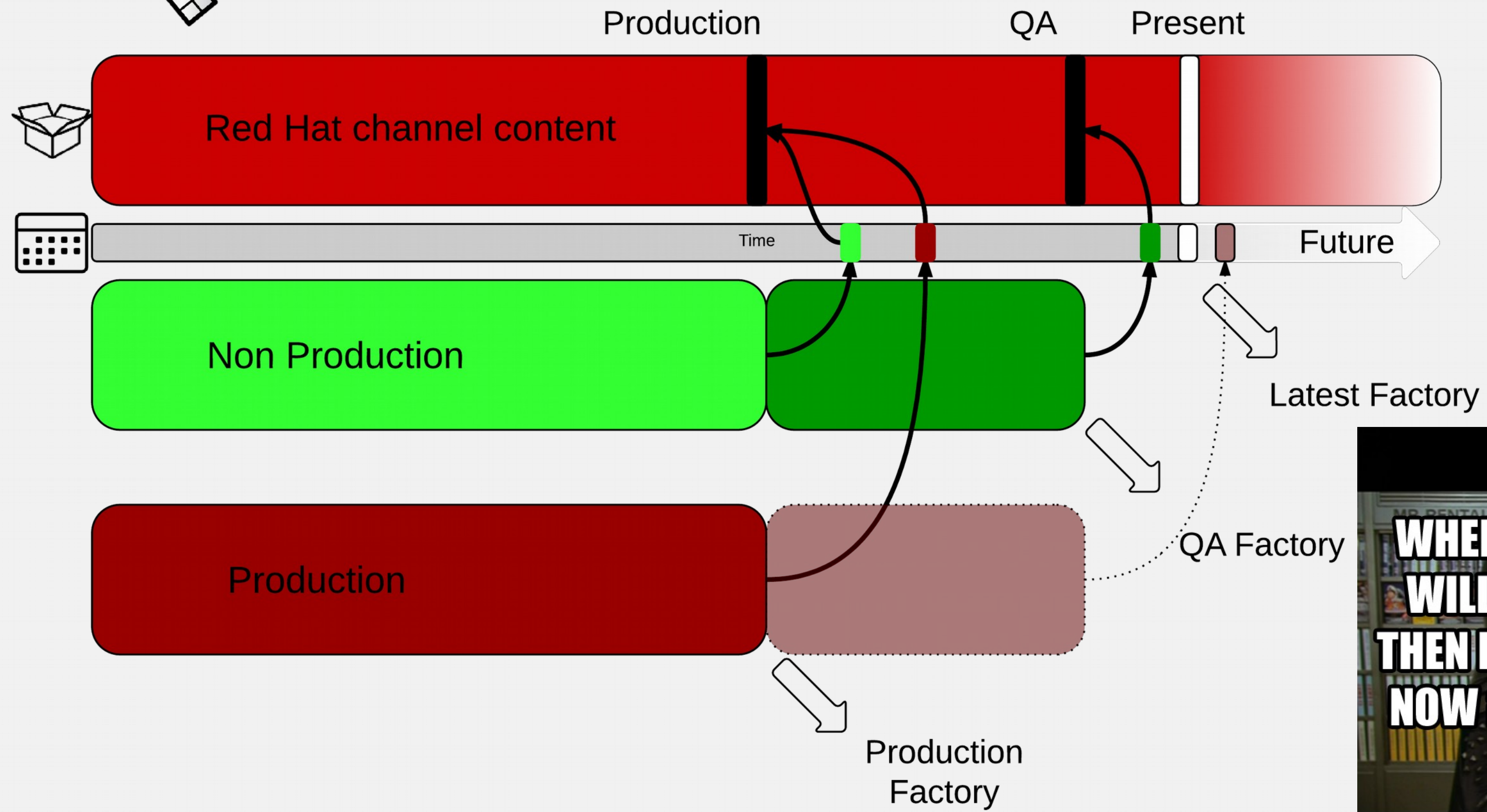


Red Hat Satellite - Manage input to factory lines



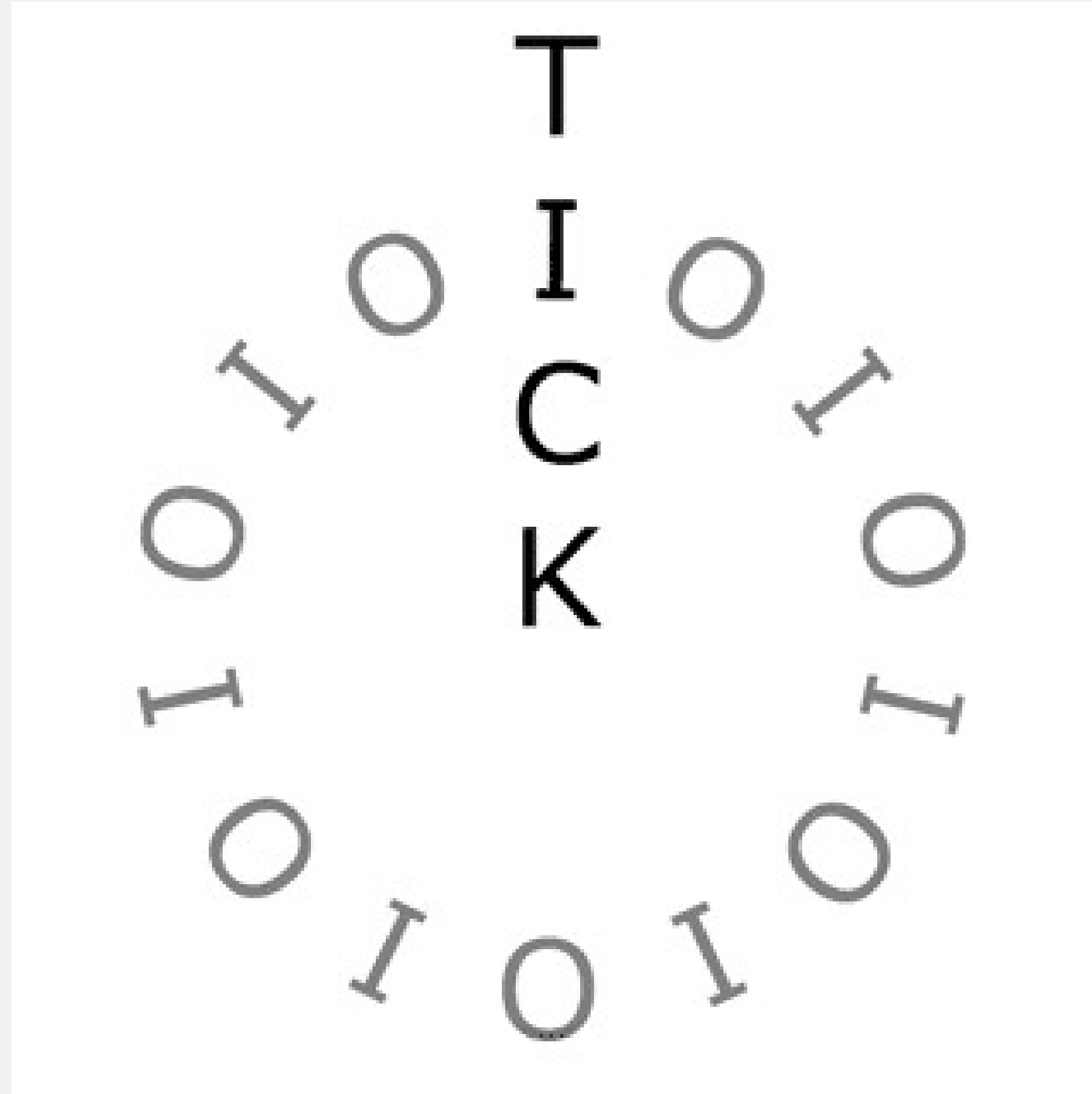


Red Hat Satellite - Manage input to factory lines

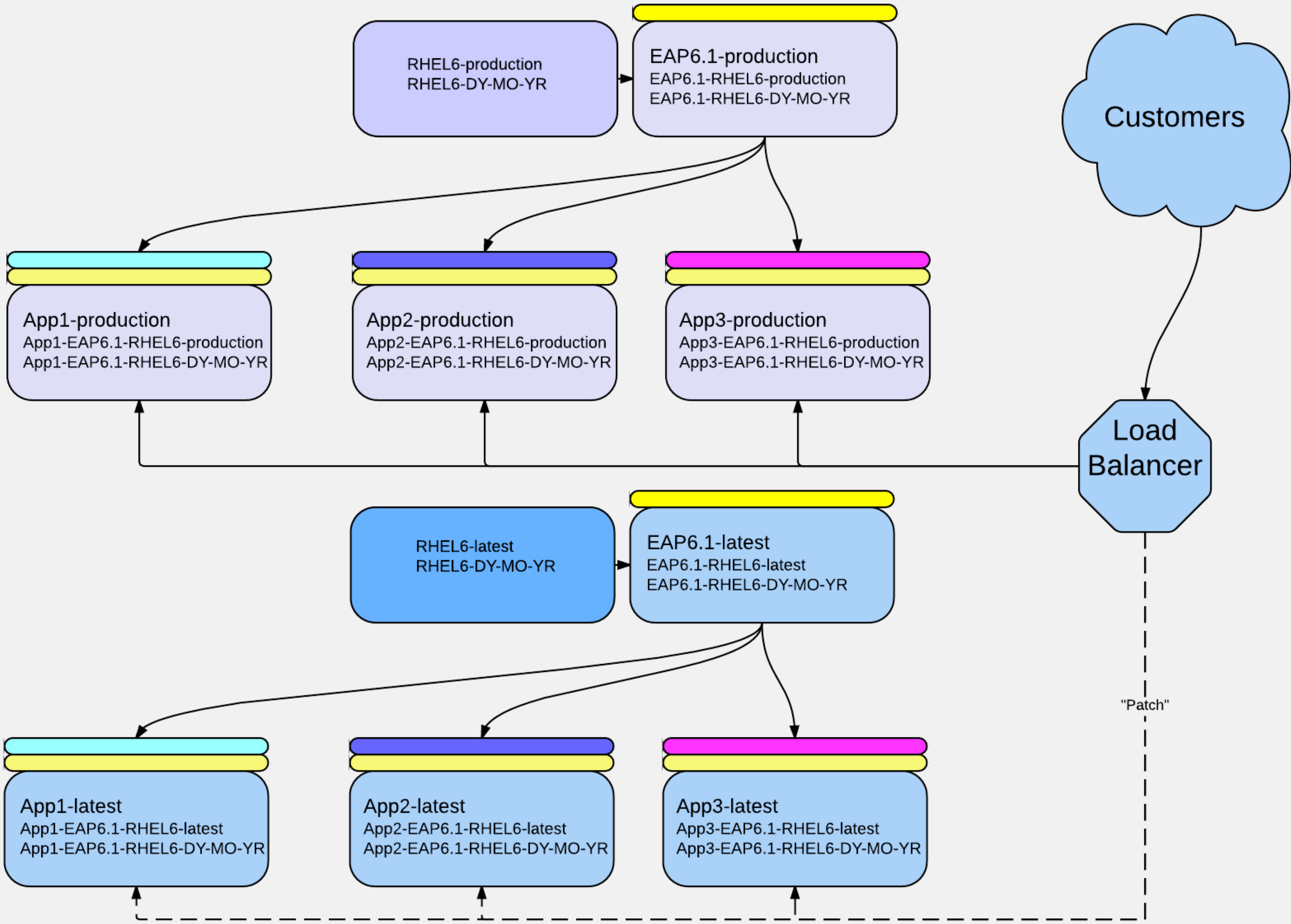


Advantages of the container factory

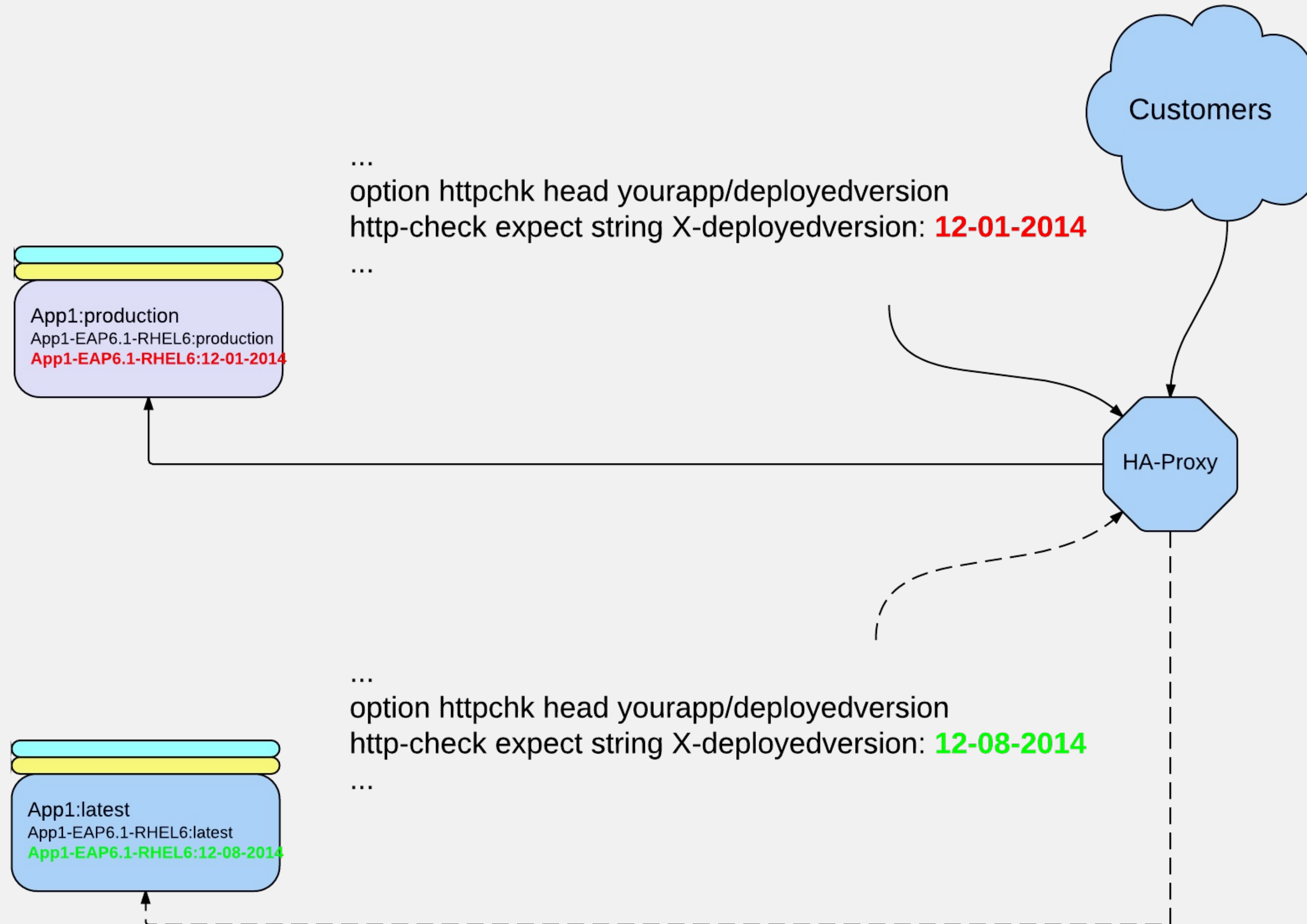
Patch cycle becomes a cutover



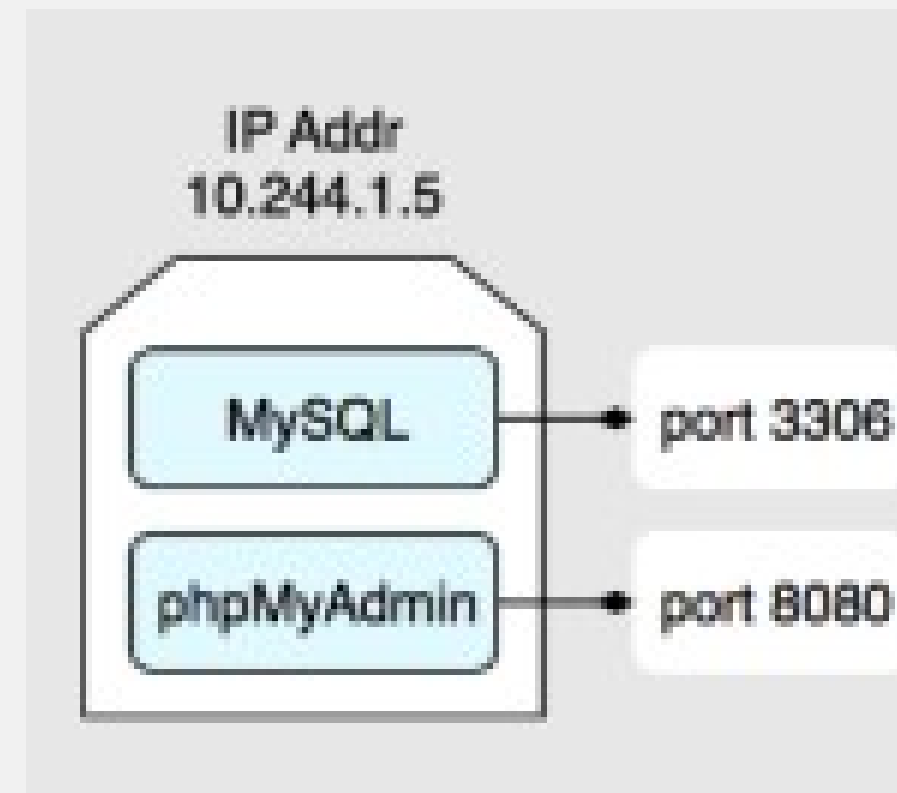
No more patching in place in a downtime window



Continuous Deployment via health checks



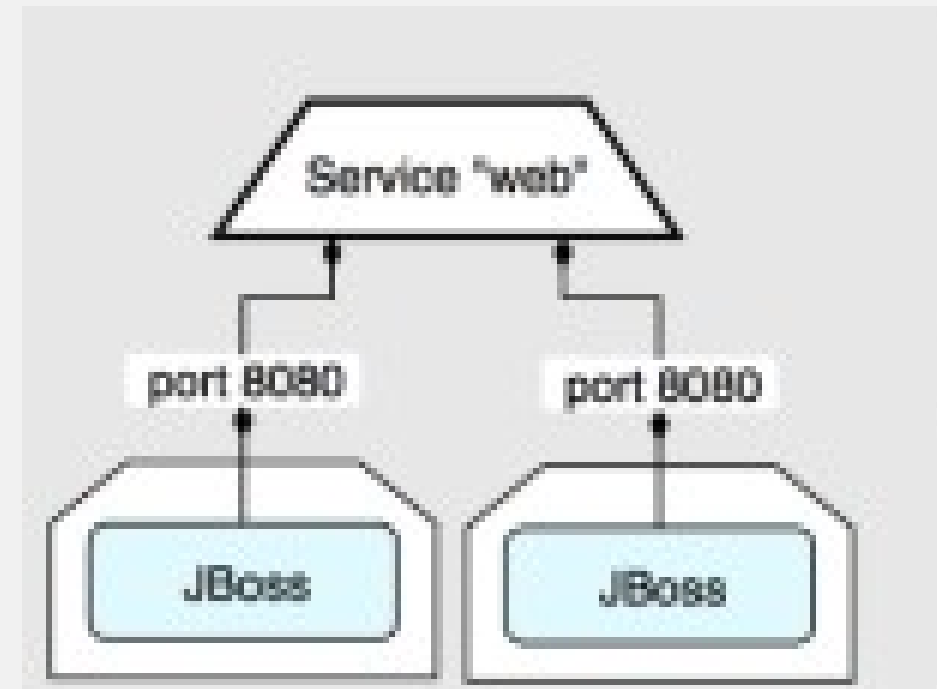
Kubernetes Concepts



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



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 rolling updates

Replication controller rolling updates - manual

- 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 production – N copies

Rolling updates - automated:

- Update the pods of frontend by just changing the image, and keeping the old name
 - `$ kubectl rolling-update yourapp --image=yourapp:v2`

OpenShift 3.0 automatic updates

Configuration Change Trigger

- The ConfigChange trigger results in a new deployment whenever changes are detected to the replication controller template of the deployment configuration.

Image Change Trigger

- The ImageChange trigger results in a new deployment whenever the value of an image stream tag changes.

Portability

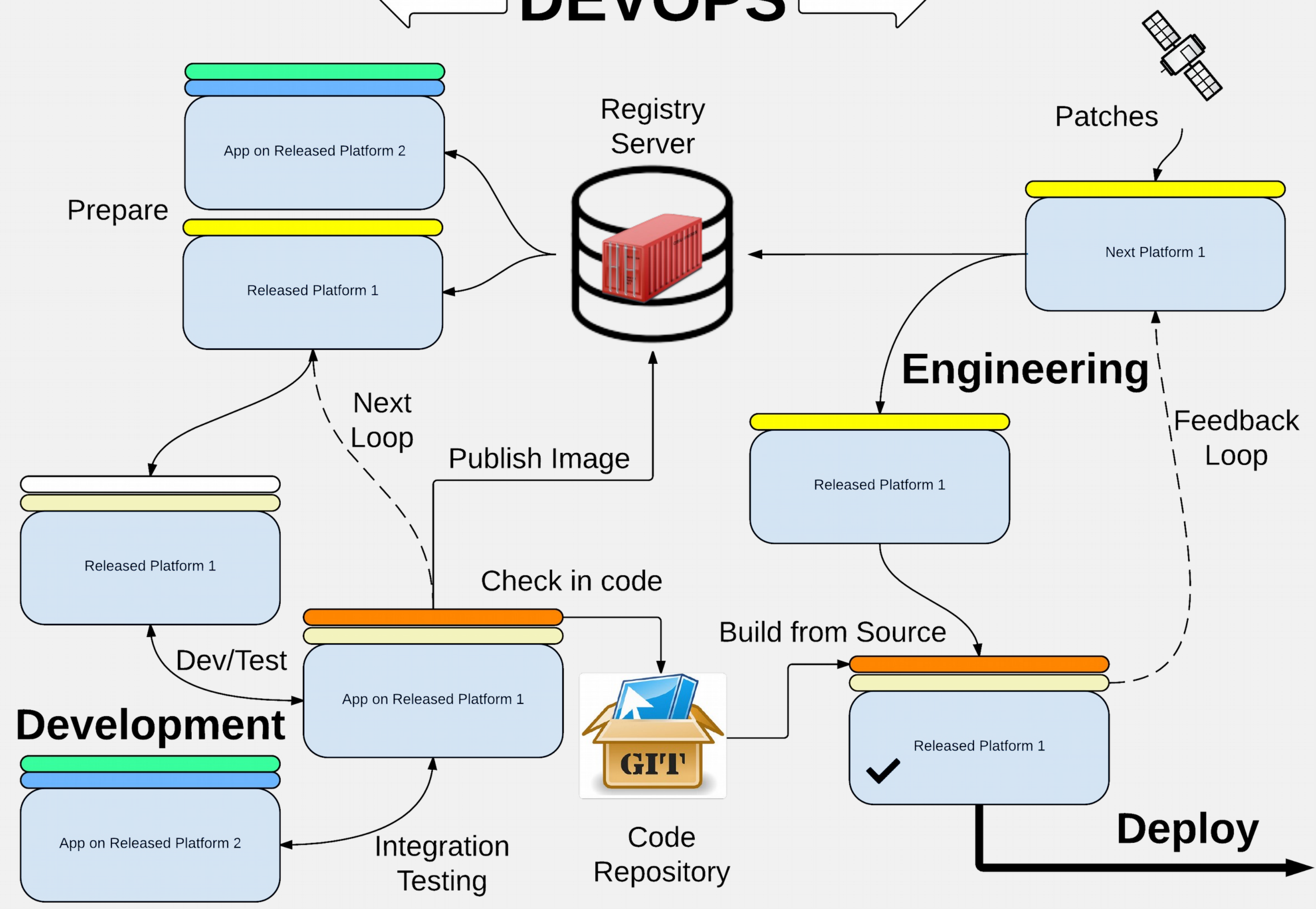
Where you build does not have to be where you run

Docker images can run anywhere RHEL can run via registry servers

- Physical
- Virtual
- OpenStack
- Public Cloud
- Developer Laptops



← DEVOPS →



Other benefits

Only Backup/Restore or make DR ready what is needed

- Jenkins server
- Satellite 6
- Source code repositories
- Databases and systems of record

Rebuild instead of Restore reduces backup load and time to recovery

- OS and Platform containers
- Build containers
- Application containers

Rapid security response capable

- Critical security patch: promote errata in Satellite 6 and rebuild production factory

Always ready to deploy

- Latest builds available for OS, Platform, Application changes
- Take images anywhere to develop or deploy, developer laptop or cloud provider

RED HAT
SUMMIT

LEARN. NETWORK.
EXPERIENCE OPEN SOURCE.