RED HAT SUMMIT

Hitachi & Red Hat collaborate: Container migration guide

In open source, we feel strongly that to do something well, you have to get a lot of people involved - Linus Torvalds

Tatsuya Yamada & Scott McCarty
Hitachi & Red Hat
May 10th, 2018

Why Hitachi & Red Hat Collaborate

Hitachi had a unique perspective on how to operationalize checklists as well as upstream Kubernetes contributions, and Red Hat had a lot of experience migrating applications in engineering and consulting.

- Collaborated on philosophy of how to tackle the problem of migrations
- Developed set of runbook like checklists around architecture, security & performance
- Published free e-Book: https://red.ht/2EkVdkJ

E-BOOK

BEST PRACTICES FOR MIGRATING TO CONTAINERIZED APPLICATIONS

BACKGROUND

When thinking about migrating applications into containers, there are three main high-level strategies: lift and shift, augment, and rewrite.[1]

No matter which method you choose, it is important to recognize that most software was designed and written before modern, image-based containers were invented.[2] Even if you choose the "lift and shift" method, where you might run a monolithic application inside a single container, it is likely that your application will need to be modified. To successfully move to containers, you will need a solid migration strategy that takes into account the needs of your applications and the nature of Linux® containers.

This document will outline specific, technical recommendations and guidelines for migrating software into containers, ranging from image build procedures to how they should run in production. The application requirements will dictate how the application should be migrated.

MINDSET

Using containers is as much of a business advantage as a technical one. When building and using containers, layering is crucial. You need to look at your application and think about each of the pieces and how they work together—similar to the way you can break up a program into a series of classes and functions. Containers are composed of packages and scripts that combine with other containers to build your application. So approach containers with the mindset that your application is made up of smaller units, and the packaging of those units into something easily consumable will make your containerized application easier to understand, deploy, and maintain.

PROPER LAYERING

The purpose of layering is to provide a thin level of abstraction above the previous layer to build something more complex. Layers are logical units where the contents are the same type of object or perform a similar task.

Basic Philosophy

Purpose & Mission

To create a piece of content that would give teams easy, but crucial technical guidance

- Make the guide operational - teams can use it day to day
- Help teams leverage their existing technical knowledge
- Add additional knowledge around how to architect applications in containers
- Highlight characteristics of containerized applications
  - Architecture
  - Performance
  - Security

Technical Guidance

Three Pillars

Breaking the problem down

Architectural

Security

Performance

Figure 1. Application requirements

Architecture

Code, Configuration, Data and more....

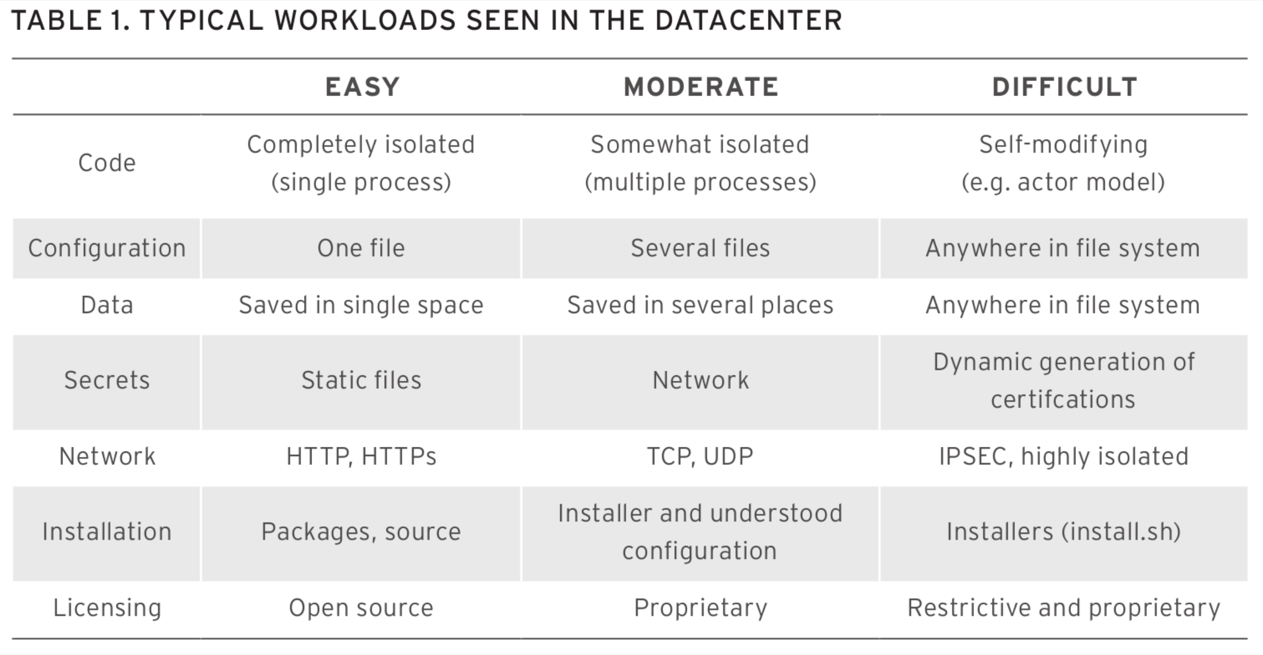


TABLE 1. TYPICAL WORKLOADS SEEN IN THE DATACENTER

	EASY	MODERATE	DIFFICULT
Code	Completely isolated (single process)	Somewhat isolated (multiple processes)	Self-modifying (e.g. actor model)
Configuration	One file	Several files	Anywhere in file system
Data	Saved in single space	Saved in several places	Anywhere in file system
Secrets	Static files	Network	Dynamic generation of certifcations
Network	HTTP, HTTPs	TCP, UDP	IPSEC, highly isolated
Installation	Packages, source	Installer and understood configuration	Installers (install.sh)
Licensing	Open source	Proprietary	Restrictive and proprietary

Performance

Virtualization & Containers are additive technologies to bare metal

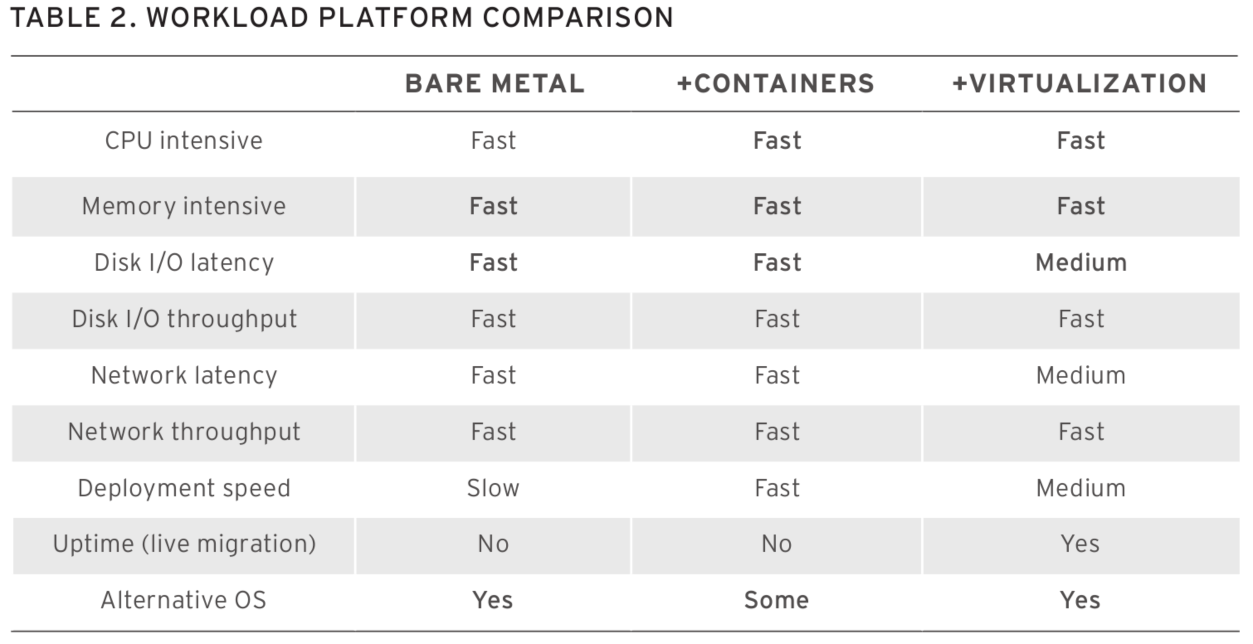


TABLE 2. WORKLOAD PLATFORM COMPARISON

	BARE METAL	+CONTAINERS	+VIRTUALIZATION
CPU intensive	Fast	**Fast**	**Fast**
Memory intensive	**Fast**	**Fast**	**Fast**
Disk I/O latency	**Fast**	**Fast**	**Medium**
Disk I/O throughput	Fast	Fast	Fast
Network latency	Fast	Fast	Medium
Network throughput	Fast	Fast	Fast
Deployment speed	Slow	Fast	Medium
Uptime (live migration)	No	No	Yes
Alternative OS	**Yes**	**Some**	**Yes**

Security

Thinking about levels of isolation....

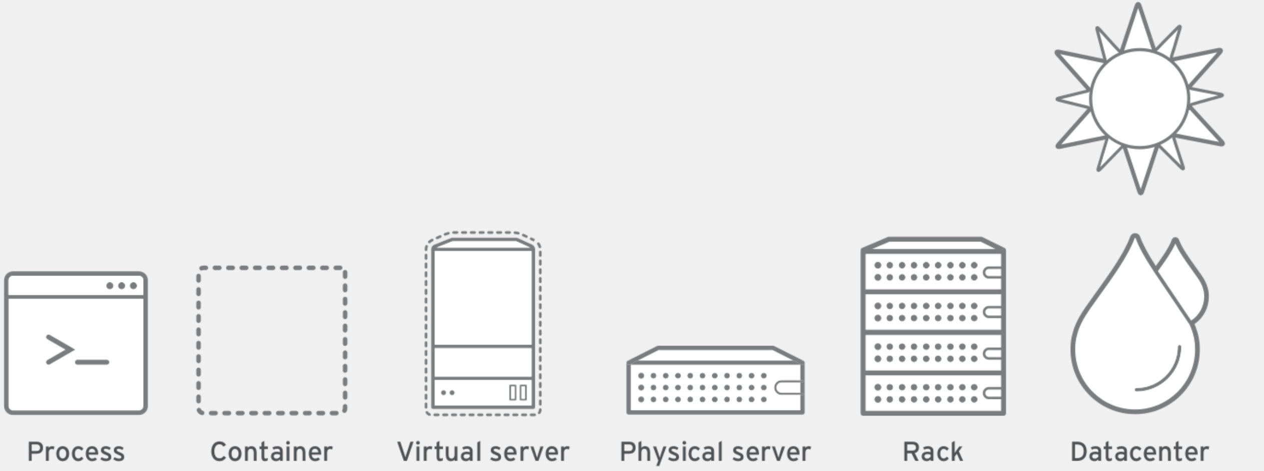
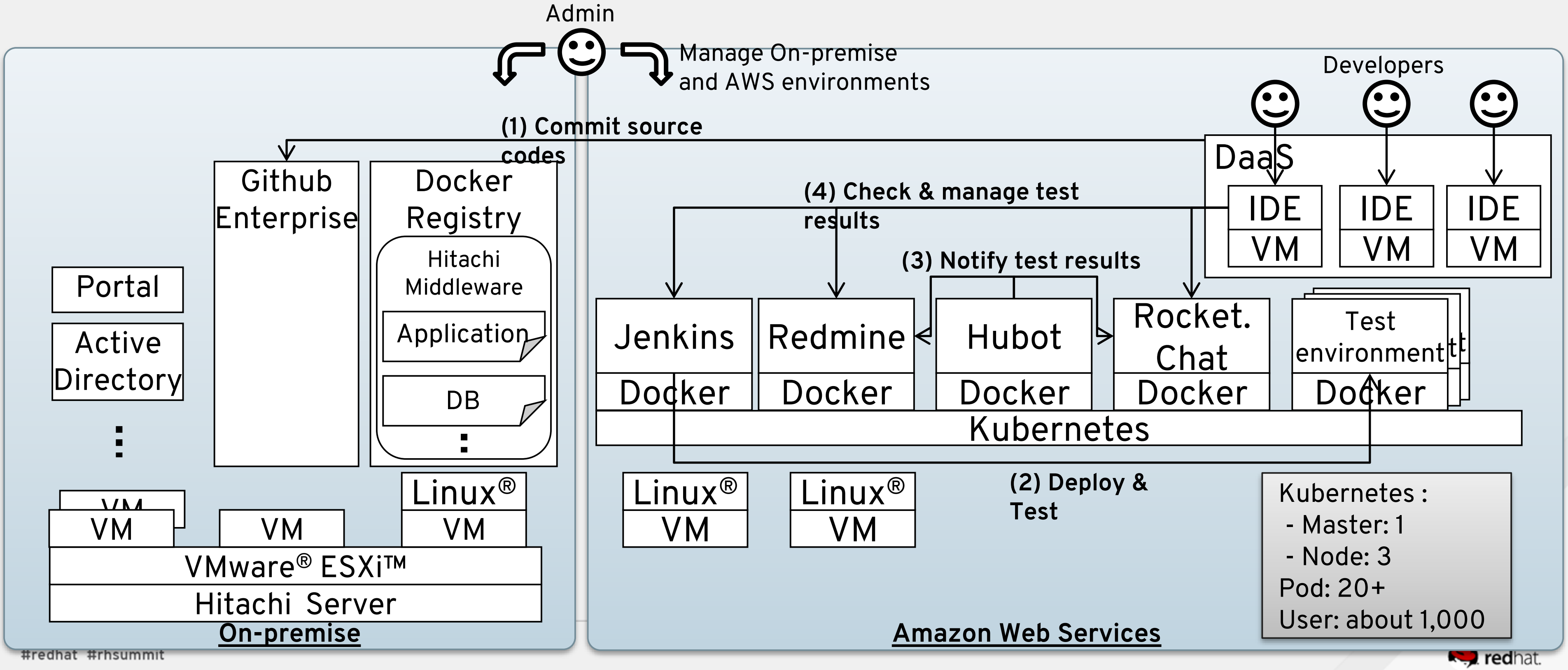


Figure 2. The tenancy scale

The Challenges with Developing Solutions

1-1. Hitachi's DevOps Approach

- Hitachi currently provides customers with following DevOps services:
  1. Hitachi's own DevOps Stack with Kubernetes and Docker
  2. OpenShift on Hitachi server

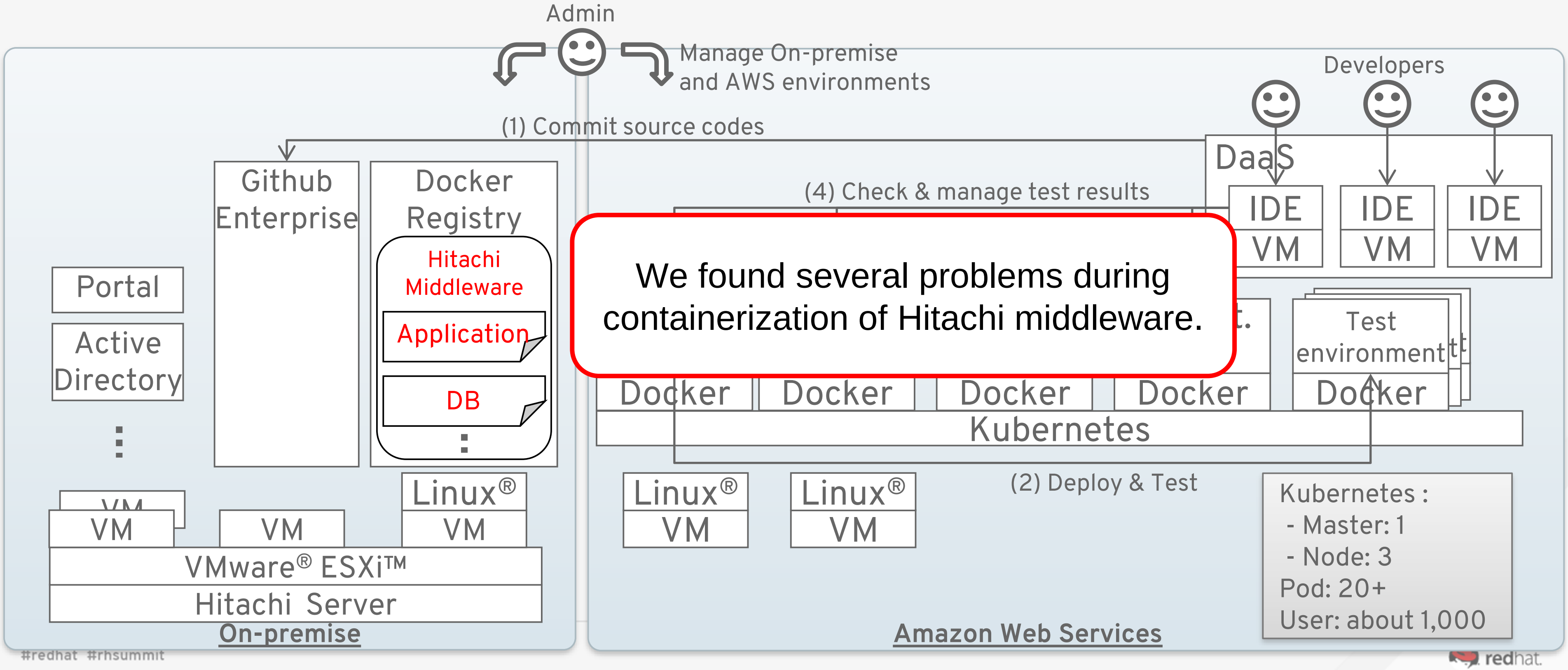


Kubernetes :
- Master: 1
- Node: 3

Pod: 20+
User: about 1,000

1-1. Hitachi’s DevOps Approach

- Hitachi currently provides customers with following DevOps services:
  1. Hitachi’s own DevOps Stack with Kubernetes and Docker
  2. OpenShift on Hitachi server

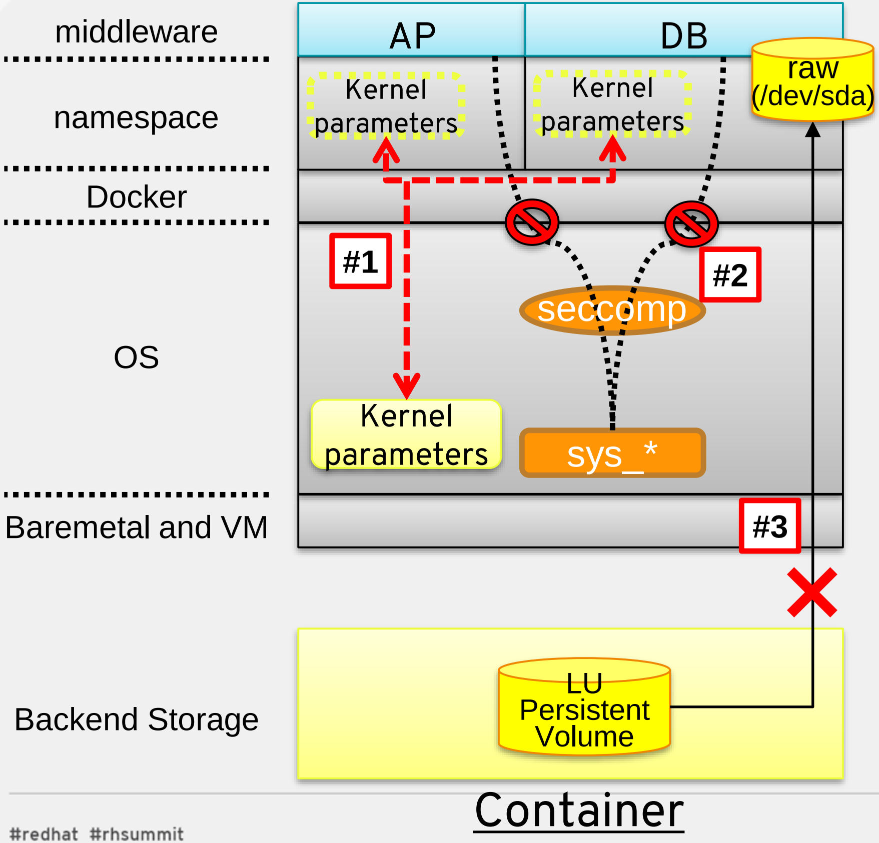


Kubernetes :
- Master: 1
- Node: 3

Pod: 20+
User: about 1,000

We found several problems during containerization of Hitachi middleware.

1-2. Prerequisite of Middleware



#1: Depending on system requirements, some middleware needs to tune kernel parameters. However, some kernel parameters cannot be configured on each container, independently.

#2: Some middleware executes system calls. For example, DB executes system call when locking a memory. However, some system calls cannot be executed on a container because it is restricted by seccomp.

#3: During containerization, customer expects that block volume can be used same as Baremetal or VM. However block volume cannot be mapped to container with Kubernetes volume.

Technical Challenges

2-1. Problems of Middleware Containerization

- The table shows 3 problems of middleware containerization we found.
- They are categorized into Performance and Security.

#	Problems	Category
1	Some kernel parameters cannot be configured on each container, independently.	Performance
2	Some system calls cannot be executed on a container.	Security
3	Block volume cannot be mapped to container with Kubernetes volume.	Performance

2-1. Problems of Middleware Containerization

- The table shows 3 problems of middleware containerization we found.
- They are categorized into Performance and Security.

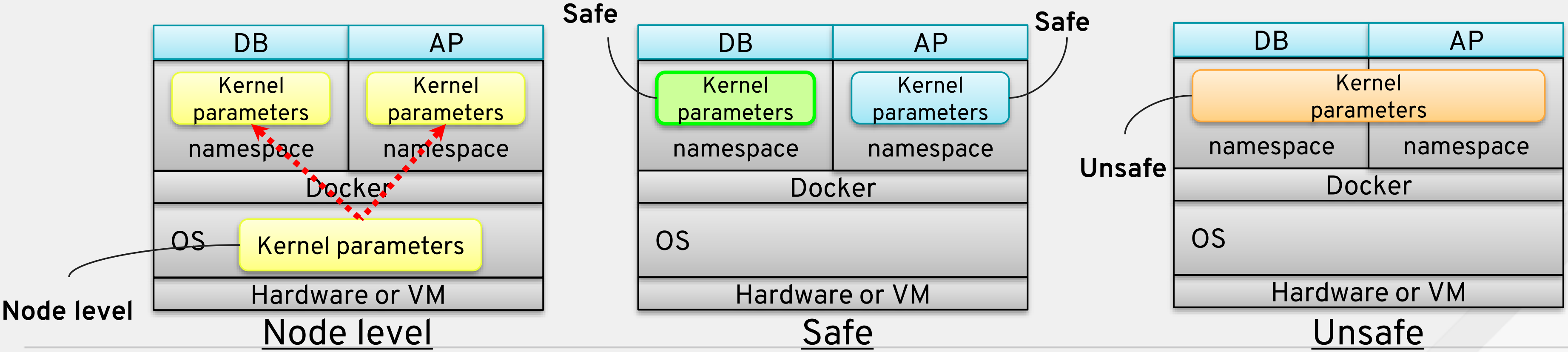
#	Problems	Category
1	Some kernel parameters cannot be configured on each container, independently.	Performance
2	Some system calls cannot be executed on a container.	Security
3	Block volume cannot be mapped to container with Kubernetes volume.	Performance

2-2. Problem of Kernel Parameters Configuration

These 3 types of kernel parameters are available for container's configuration.

Parameters	Range	Detail
Node level	Node	This can be set for each node, but can not be set for each container.
Safe	Pod	This can be set for each container, and does not affect other container.
Unsafe	Pod	This can be set for each container, but may affect other containers.

- Containerized middleware like DB needs to set kernel parameters even if it's in container.
- However setting Node level sysctls or unsafe.sysctls may affect another containers.

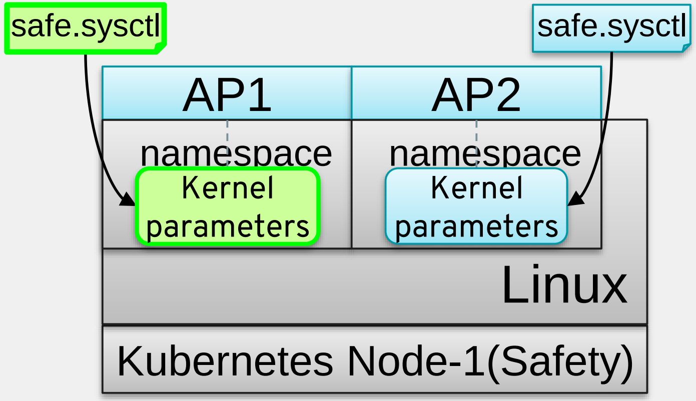


- OpenShift Container Platform 3.9 - sysctls
  https://access.redhat.com/documentation/en-us/openshift_container_platform/3.9/html/cluster_administration/admin-guide-sysctls

2-3. Solution of Kernel Parameters Configuration

Place Pod with safe.sysctl on “Safety” node.

Safe.sysctl can configure each Pods without influence of unsafe and Node level.

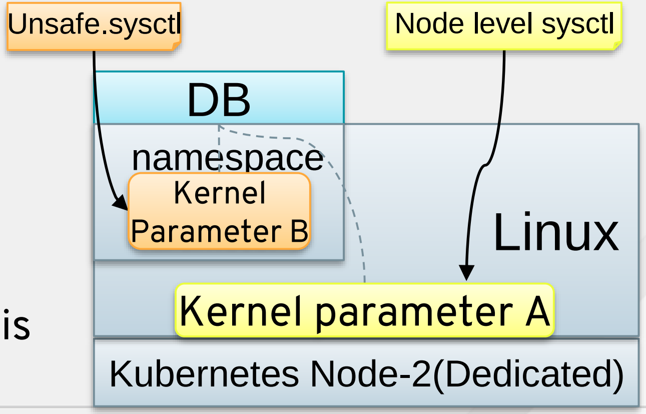


Place Pods with unsafe.sysctl or Node level sysctl on “Dedicated” node.

a. Configure “Kubernetes Taints” to dedicated nodes beforehand, so that only specific Pods can be placed on “Dedicated” node.

b. Set sysctl settings to the “Dedicated” nodes.

c. Create a Pod with “Kubernetes Tolerate” so that the pod is placed on the Taint Node like Kubernetes Node-2.

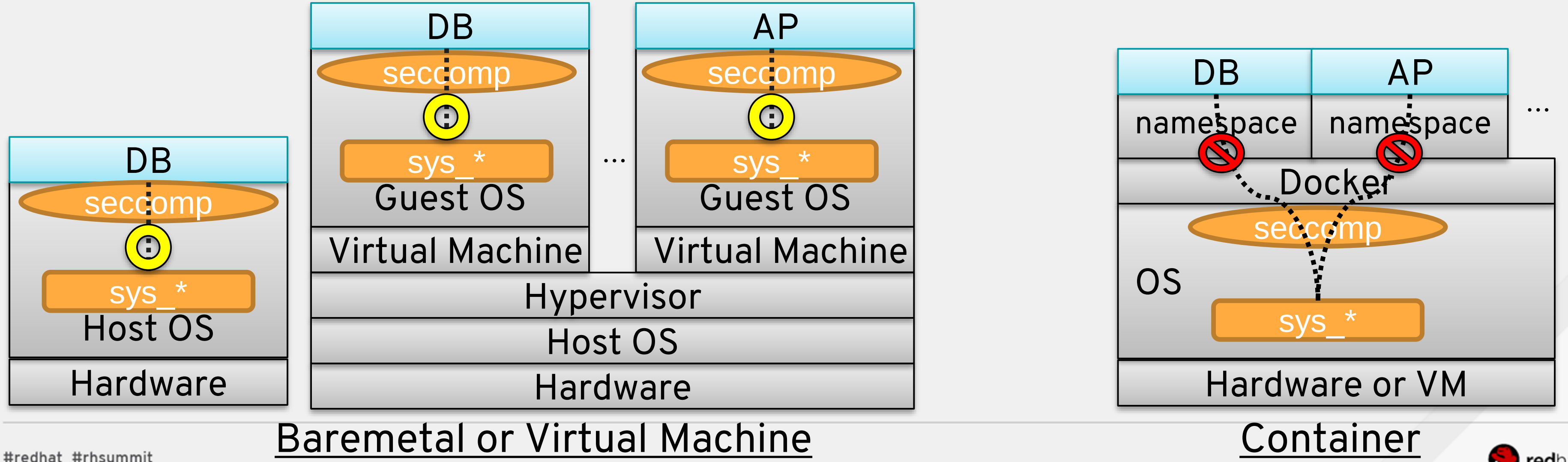


2-4. Problem of Middleware Containerization

#	Problems	Category
1	Some kernel parameters cannot be configured on each container, independently.	Performance
2	Some system calls cannot be executed on a container.	Security
3	Block volume cannot be mapped to container with Kubernetes volume.	Performance

2-5. Problem of System Calls

- Inside a container, some system calls are restricted by default.
- As a result, operations of a container application are restricted.
  - Ex. Core Dump
    Container cannot issue “ulimit” command. Therefore core dump of the application doesn’t get dumped.

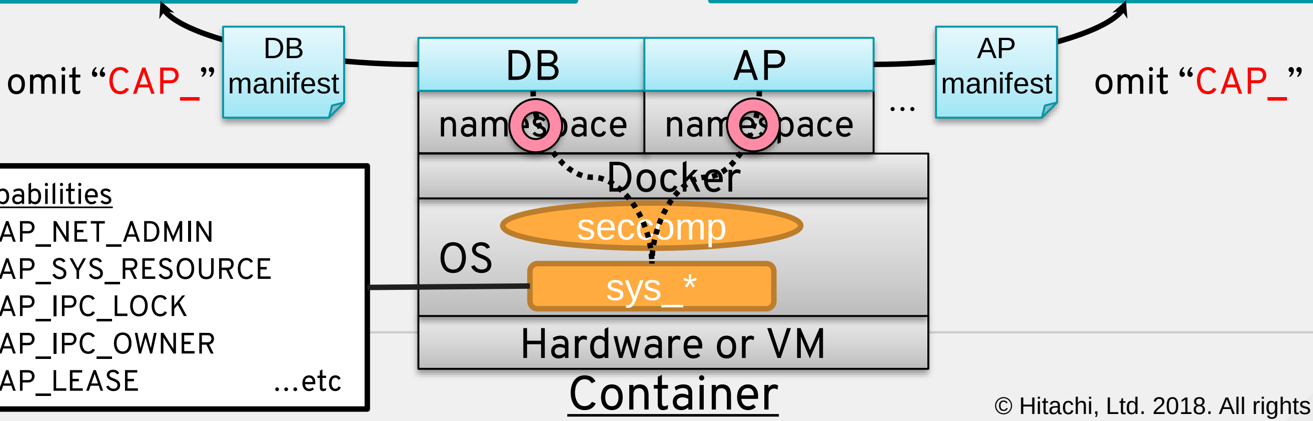


2-6. Solution of System Calls

- To make system call executable,
  1. Set “seccomp=unconfined”
  2. Add specified Linux Capabilities.

```
seccomp.security.alpha.kubernetes.io/pod: unconfined
securityContext:
  capabilities:
    add:
      - NET_ADMIN
      - SYS_RESOURCE
      - IPC_LOCK
      - IPC_OWNER
      - LEASE
```

```
seccomp.security.alpha.kubernetes.io/pod: unconfined
securityContext:
  capabilities:
    add:
      - NET_ADMIN
      - SYS_RESOURCE
      - IPC_LOCK
      - IPC_OWNER
```



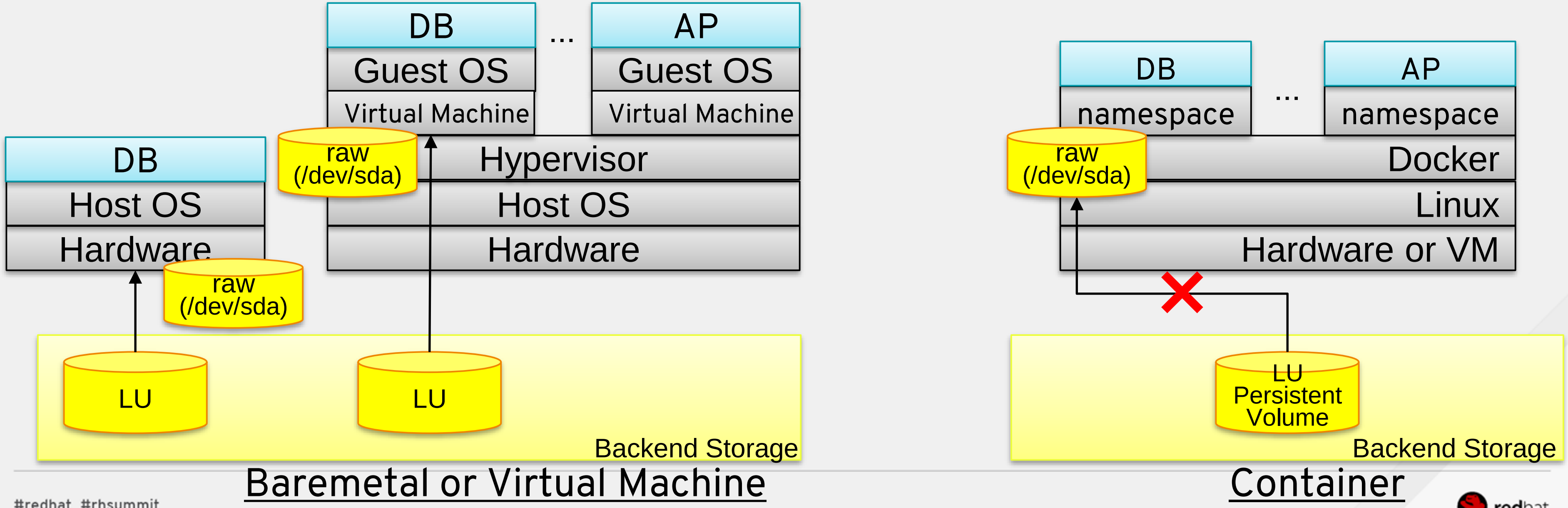
Capabilities
- CAP_NET_ADMIN
- CAP_SYS_RESOURCE
- CAP_IPC_LOCK
- CAP_IPC_OWNER
- CAP_LEASE ...etc

2-7. Problems of Middleware Containerization

#	Problems	Category
1	Some kernel parameters cannot be configured on each container, independently.	Performance
2	Some system calls cannot be executed on a container.	Security
3	Block volume cannot be mapped to container with Kubernetes volume.	Performance

2-8. Problem of Block Volume support

- During containerization, customer expects that block volume can be used same as Baremetal or Virtual Machine because block volume provides consistent I/O performance and low latency compared to filesystem volume.
- However, before version 1.8, Kubernetes didn't support attaching block volume to container from backend storage.



2-9. Solution of Block Volume support

- Hitachi developed a feature that enables to use block volume support in cooperation with Red Hat at Kubernetes community.
- This feature enables to attach block volume directly to the container.
- New parameters “volumeMode”, “volumeDevices” and “devicePath” were added to configure block volume support. Ex: volumeMode=“Filesystem” or “Block”.

Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: block-pv001
spec:
 capacity:
  storage: 1Gi
 accessModes:
  - ReadWriteOnce
 volumeMode: Block
 persistentVolumeReclaimPolicy: Retain
 fc:
  targetWWNs: [‘28000001ff0414e2‘]
  lun: 0
```

Persistent Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: block-pvc001
spec:
 accessModes:
  - ReadWriteOnce
 volumeMode: Block
 resources:
  requests:
   storage: 1Gi
```

Pod

```
apiVersion: v1
kind: Pod
metadata:
 name: blockvolume-pod
spec:
 containers:
  - name: blockvolume-container
      ...
    volumeDevices:
     - name: data
       devicePath: /dev/xvda
 volumes:
  - name: data
   persistentVolumeClaim:
    claimName: block-pvc001
    readOnly: false
```

2-9. Solution of Block Volume support

- Hitachi developed a feature that enables to use block volume support in cooperation with Red Hat at Kubernetes community.
- This feature enables to attach block volume directly to the container.
- New parameters “volumeMode”, “volumeDevices” and “devicePath” were added to configure block vol

Persisten

```
apiVersion:
kind: Persis
metadata:
 name: bloc
spec:
 capacity:
  storage: 1(
 accessMod
  - ReadWrite
 volumeMode: Block
 persistentVolumeReclaimPolicy: Retain
 fc:
  targetWWNs: [‘28000001ff0414e2‘]
  lun: 0
```

```
  requests:
   storage: 1Gi
```

```
ntainer

  - name: data
   devicePath: /dev/xvda
volumes:
 - name: data
  persistentVolumeClaim:
   claimName: block-pvc001
   readOnly: false
```

Future Work

- Block volume is Alpha version in Kubernetes v1.10, therefore more unit tests and e2e test cases are required to improve reliability.

Solution Summary

3. Conclusion

#	Problems	Our approach
1	Some kernel parameters cannot be configured on each container, independently.	Pros: Divide node for each purpose. Cons: Container hosts are increase.
2	Some system calls cannot be executed on a container.	Pros: Use seccomp and Linux Capabilities for configuration of each application container.
3	Block volume cannot be mapped to container with Kubernetes volume.	Pros: Block volume support were merged at Kubernetes v1.9.

Learn How to Migrate

Check Out the Guide

Download the e-Book:

https://red.ht/2EkVdkJ

E-BOOK

BEST PRACTICES FOR MIGRATING TO CONTAINERIZED APPLICATIONS

BACKGROUND

When thinking about migrating applications into containers, there are three main high-level strategies: lift and shift, augment, and rewrite.[1]

No matter which method you choose, it is important to recognize that most software was designed and written before modern, image-based containers were invented.[2] Even if you choose the "lift and shift" method, where you might run a monolithic application inside a single container, it is likely that your application will need to be modified. To successfully move to containers, you will need a solid migration strategy that takes into account the needs of your applications and the nature of Linux® containers.

This document will outline specific, technical recommendations and guidelines for migrating software into containers, ranging from image build procedures to how they should run in production. The application requirements will dictate how the application should be migrated.

MINDSET

Using containers is as much of a business advantage as a technical one. When building and using containers, layering is crucial. You need to look at your application and think about each of the pieces and how they work together—similar to the way you can break up a program into a series of classes and functions. Containers are composed of packages and scripts that combine with other containers to build your application. So approach containers with the mindset that your application is made up of smaller units, and the packaging of those units into something easily consumable will make your containerized application easier to understand, deploy, and maintain.

PROPER LAYERING

The purpose of layering is to provide a thin level of abstraction above the previous layer to build something more complex. Layers are logical units where the contents are the same type of object or perform a similar task.

RED HAT SUMMIT

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHat

youtube.com/user/RedHatVideos

Trademarks

- HITACHI is a registered trademark of Hitachi, Ltd.
- Red Hat and OpenShift are trademarks or a registered trademarks of Red Hat Inc. in the United States and other countries.
- Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
- Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries.
- Kubernetes is a registered trademark of The Linux Foundation.
- Amazon Web Services is a trademark of Amazon.com, Inc. or its affiliates in the United States and/or other countries.
- VMware, ESXi are registered trademarks or trademarks of VMware, Inc. in the United States and other jurisdictions.
- Active Directory is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.
- Other company and product names mentioned in this document may be the trademarks of their respective owners.