Presenter: Veer Muchandi
Title: Chief Architect - Container Solutions
Social Handle: @VeerMuchandi
Blogs: https://blog.openshift.com/author/veermuchandi/
Container Technologies - Impact on Enterprises

Container Strategy is on every CTO’s agenda. You may be already using containers or just starting off your journey in your enterprise. This session provides you an executive level understanding of how the technologies are evolving in the Container Landscape.

Will walk step by step through container technologies:

- At the Operating System level
- Container Orchestration
- Building Containerized Applications
- Managing your containerized microservices
- Service Mesh
- Serverless with Containers and more..

Will cover the use cases for these technologies, and when to think about applying these technologies in your enterprise. We will also understand how the cloud providers are approaching these areas, the available open source options and the reasons to make such choices.

All in all, if you are a CxO, an IT Executive, an IT Manager or someone who wants to have paint a picture of container technology landscape in your mind, you must attend this session.
Goals

Executive level understanding of Container Landscape
- Containers and relationship to Microservices
- Containers and impact on CICD
- Container Architecture and related standards
- Container Orchestration/Kubernetes and related standards
- Service Mesh Technology Intro
- Serverless Technologies Intro
Containers - Transform Apps, Infrastructure & Process

Application Architecture
- Monolithic
- N-Tier
- Microservices

Development Process
- Waterfall
- Agile
- DevOps

Application Infrastructure
- Datacenter
- Hosted
- Cloud

Containers
Enable efficiency and automation for microservices, but also support traditional applications

Enable faster and more consistent deployments from Development to Production

Enable application portability across 4 infrastructure footprints: Physical, Virtual, Private & Public Cloud
Understanding Containers
What Are Containers?

It Depends on Who You Ask

- Sandboxed application processes on a shared Linux OS kernel
- Simpler, lighter, and denser than virtual machines
- Portable across different environments

INFRASTRUCTURE

APPLICATIONS

- Package my application and all of its dependencies
- Deploy to any environment in seconds and enable CI/CD
- Easily access and share containerized components
Understanding Containers

Virtual Machines vs Containers
Virtualization vs. Containers

Virtualization

- Virtual Machine
  - User Space
  - Kernel Space
  - Hypervisor

Layer of Abstraction

Containerization

- Container
  - User Space
  - Kernel Space
  - Hypervisor

Layer of Abstraction
Microservices and Containers
A Typical Monolith

Multiple business functions all bundled up into a single large monolith

Acknowledgements: Borrowed a few conventions from here http://martinfowler.com/articles/microservices.html
Deployed as a single large deployment unit on the host.
- hard to change
- hard to manage
- causes slow cycles

So we want to break it up..
So we want to refactor each business function as an independent microservice.

So how does a microservice run on a host?
Microservices are typically very small.

Even the smallest sized VM in your enterprise may be too big.

with a single microservice per host, we will end up wasting a lot of resources.

So should we run a bunch of them on the box?
Well.. now all our eggs are in the same basket!!

Hmm.. let's see.

How about? .....
Mixing different microservices on a host.

Can I do this?
wait.. Microservices are Polyglot.

Each language has its own libraries, dependencies.

Phew.. how do we deal with this on one host?
Multiple containers run on a host.

Containers share kernel on the Host

Containers have layers infrastructure-as-code by default!
It is not just your application but all dependencies included.
Container Layers and Responsibilities

- Core Build
  - Core Build (Dockerfile)
  - Core Build (Dockerfile)
  - Core Build (Dockerfile)

- Middleware
  - Middleware (Dockerfile)
  - Middleware (Dockerfile)
  - Middleware (Dockerfile)

- Application
  - Application Code (Dockerfile)
  - Application Code (Dockerfile)
  - Application Code (Dockerfile)
Build Automation
Again multiple containers run on a host.

Since dependencies are bundled in, each container can implement its own technologies, without affecting other containers on the host.

Containers are naturally Polyglot.
Speed of Deployment

VMs take minutes to come up

Container spin up fast .. in seconds!!!
Containers Scale up fast and Scale down fast

Microservices need to scale up quickly. Containers provide that OOTB.
Application Upgrades, Security fixes, Middleware, BaseOS Upgrades

Upgrades are quick

They won't affect other containers as the changes are local to a container

Easy to rollback.. Just bring up the previous container version!!

Meets this need...
"Microservices can be changed quickly without affecting others!!"
Containers and CI/CD
Pipelines with Containers

Traditionally we hand over application code (eg: Jar) across the environment.

With Container based model, the artifact the passes across environments is a Container Image that includes application and its dependent libraries.
Containers are Portable

Containers run the same way across the datacenters. Portability comes with the container format.

So do you want to burst your microservices to other datacenters or cloud to meet your demands??
Architecture - Container Engine, Runtime, Standards
Containers are Linux (or Windows) processes
The Internet is WRONG :-)

Important corrections

- Containers do not run ON docker. Containers are processes - they run on the Linux kernel. Containers are Linux processes (or Windows).
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers
Container Engines

Podman
- Podman
- runc

CRI-O
- CRI-O
- runc

Docker
- dockerd
- containerd
- runc
Containers and Non-Container processes run on the Host

How do containers run on the host?
Container Engine provides necessary functionality

Container Engine:
- Container Runtime. Default: runc
- API for user interaction
- Image Push/Pull from Registry
- Download Image layers
- Prepare Storage
- Extract Image
- Prepare configuration
- Handover to Container Runtime
- Container Runtime calls the kernel
- Kernel creates a Container
  - Namespaces
  - SELinux & MCS
  - Linux Control Groups
- Container runs on Kernel
Standards Matter

Protect customer investment
The world of containers is moving very quickly. Protect your investment in training, software, and building infrastructure.

Enable ecosystems of products and tools
Cloud providers, software providers, communities and individual contributors can all build tools.

Allow communities with competing interests to work together
There are many competing interests, but as a community we have common goals.
Container Standards

The building blocks of how a container goes from image to running process

- **A15B7FF8.json**
  - Image Layers
  - OCI Image Specification
  - Allows users to build container images with any tool they choose. Different tools are good for different use cases.

- **config.json**
  - Root Filesystem
  - Container Engine
  - The container engine is responsible for creating the config.json file and unpacking images into a root file system.

- **config.json**
  - Root Filesystem
  - OCI Runtime Specification
  - OCI compliant runtimes can consume the config.json and root filesystem, and tell the kernel to create a container.

- **clone(childFunc)**
  - Operating System
  - OCI compliant runtimes can be built for multiple operating systems including Linux, Windows, and Solaris
Eco-system for Containers
Podman/Buildah in place of Docker

Daemon-less, Non-root
Container Orchestration
Container Orchestration and Management is about

- Running containerized applications on a cluster at enterprise scale
- Deciding the right host for your container with enough resources
- Ensuring containerized application high availability - resilience and recovery from failure
- Scaling containers up and down based on workload needs
Popular Orchestration Technologies

- Kubernetes
- Cloud Foundry
- Docker Swarm
- Mesos
Kubernetes Adoption

Google Kubernetes Engine

Red Hat OpenShift

TECTONIC by CoreOS

Early Adopters

And many more eventual implementations..
So let’s focus on Kubernetes as a Container Orchestrator
Openshift/K8S runs containers in Pods. Pod is a wrapper

Each pod gets an IP address. Container adopts Pod's IP.
Some pods may have more than one container... that's a special case though!!

Usually these containers are dependent like a master and slave or side-car pattern.

And they have a very tight married relationship.

All the containers in a pod die along with a pod.
Pod Scaling

When you scale up your application, you are scaling up pods.

Each Pod has its own IP.
Nodes are the application hosts that make up an OpenShift/K8S cluster. They run Docker and Openshift. The Master controls where the pods are deployed on the nodes, and ensures cluster health.
When you scale up, pods are distributed across nodes following scheduler policies defined by the administrator. So even if a node fails, the application is still available.
Health Management

Not just that, if a pod dies for some reason, another pod will come in its place.
Flexibility of architecture with Openshift/ K8S Services

Pods can be front-ended by a Service. Service is a proxy. Every node knows about it. Service gets an IP.

Service knows which pods to front-end based on the labels.
Clients can talk to the service. Service redirects the requests to the pods.

Service also gets a DNS Name

Client can discover service... built in service discovery!!
Accessing your Application

When you want to expose a service externally eg: access via browser using a URL, you create a "Route". Route gets added to a HAProxy LB.

You can configure your F5 as well as LB.
Additional Container Standards

- Open Containers Initiative (OCI) Image Specification
- Open Containers Initiative (OCI) Distribution Specification
- Open Containers Initiative (OCI) Runtime Specification
- Container Runtime Interface (CRI)
- Container Network Interface (CNI)

Standard for Container Runtimes to work with Kubernetes
Standard for Network providers to work with Kubernetes
Kubernetes Standards

What does Kubernetes need?

- A standard way for kubelet to communicate with the Container Engine
  - **Container Runtime Interface (CRI)**: A protocol between Kubelet and Engine
- A Daemon with communicates with CRI
  - **gRPC server**: a daemon or shim which implements this spec
- A standard way for humans to interface with gRPC server for troubleshooting/debugging
  - **cri-ctl**: a node based CLI tool that can list images, view running containers etc.
cri-o for Kubernetes

CRI-O is an implementation of CRI spec
- Production Ready
- Optimized for Kubernetes
- Pull from any OCI compliant registry
- Run any OCI compliant container
Understanding Service Mesh
Calls with Distributed Computing
World of Microservices

100s of Microservices running on cluster

Issues to address:
- Load balancing
- Network Faults
- Circuit Breakers
- Service Discovery
- QOS & SLA
and more..

Requires “Plumbing Code”
Current Solutions

BL - Business Logic

PL - Plumbing Logic to solve the infrastructure issues such as Service discovery, load balancing, fault-tolerance, rate limiting, QoS etc.

Examples: Frameworks such as NetflixOSS
Issues with libraries/frameworks embedded in code

Developers have to worry about the plumbing code

- Code intrusive
- Learning curve for such frameworks
- Not language-agnostic; hampers polyglot microservices
- Maintenance overhead
  - thousands of services using version of libraries
  - updates to infra libraries require integration, testing and re-deployment of all services
While Kubernetes goes to an extent in terms of the handling infrastructure needs such as service discovery, load balancing and some request routing, we need more OOTB features -

- Content based routing
- Canary, AB deployments
- Rate Limiting
- End-to-End access control
- Fault tolerance and fault injection
- Routing, Ingress and Egress rules
- Circuit Breakers
- Integration with tools for logging, monitoring, quotas, ACLs and more
**Sidecar Proxy**

- Intercepts all network communication between microservices
- Encapsulates Service Infrastructure code
- Application code (business logic) unaware of Sidecar proxy
- Examples - Linkerd, Envoy
Service Mesh

Network of Microservices

Service Mesh is a dedicated infrastructure layer to handle service-service communications.

Typically implemented as an array of lightweight network proxies deployed alongside application code.

Interconnected Proxies form a mesh network.
Implementation of mesh network involves a “Control Plane”

Proxies managed by centralized Control Plane
Istio Service Mesh

HTTP/1.1, HTTP/2, gRPC or TCP -- with or without mTLS

Policy checks, telemetry

Configuration data to proxies

Configuration data

TLS certificates to proxies
Serverless Solutions with Containers
Serverless is not really “server”-less

It is about consumer (deployer) of a service **not worrying about**:

- How to provision/de-provision/manage infrastructure to run the service?
- Application scaling up/down based on workload needs
- Idle services using resources when not needed

**FaaS**: Compose application as individual autonomous functions. Calls functions based on events. Run as serverless model
Bringing serverless model to enterprise infrastructure

Knative provides primitives to implement Serverless Model on enterprise infrastructure

- Runs as containers
- Builds on Kubernetes and Service Mesh
- Aiming to become a defacto standard for serverless
## Knative Features

<table>
<thead>
<tr>
<th>Build</th>
<th>Serving</th>
<th>Eventing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides easy-to-use, simple source-to-container builds, so you can focus on writing code and know how to build it. Knative solves for the common challenges of building containers and runs it on cluster.</td>
<td>Run serverless containers on Kubernetes with ease, Knative takes care of the details of networking, autoscaling (even to zero), and revision tracking. You just have to focus on your core logic.</td>
<td>Universal subscription, delivery, and management of events. Build modern apps by attaching compute to a data stream with declarative event connectivity and developer-friendly object model.</td>
</tr>
</tbody>
</table>
Knative for Serverless (anywhere)
Red Hat OpenShift Container Platform

Service

OpenShift Cloud Functions
(Knative+ Keda)

OpenShift Service Mesh
(Istio + Jaeger + Kiali)

OpenShift Container Platform
(Enterprise Kubernetes)

Serverless

Infra Ops

Infra

FUNCTIONS, APPLICATIONS, MICROSERVICES

ANY INFRASTRUCTURE

INFRA

Laptop

Datacenter

OpenStack

Amazon Web Services

Microsoft Azure

Google Cloud
THANK YOU

+ plus.google.com/+RedHat

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHatNews

twitter.com/RedHatAtomic
Container Standards

- Open Containers Initiative (OCI) Image Specification
- Open Containers Initiative (OCI) Distribution Specification
- Open Containers Initiative (OCI) Runtime Specification
- Container Runtime Interface (CRI)
- Container Network Interface (CNI)