A Greybeard's Worst Nightmare

How Kubernetes and Containers are re-defining the Linux OS

Daniel Riek, Sr. Director, Systems Design & Engineering, Red Hat
About me

• Name: Daniel Riek   Twitter: llunved
• Using Linux since 1994
• Co-founded Linux start-up ID-Pro in Europe in 1997
• Worked at Alcove, a french GNU/Linux company 2001-2003
• Red Hat, EMEA Sales Engineering 2003-2005
• Red Hat, PM for RHEL 2005-2011
• CTO at trading startup Vincorex 2011-2012
• PM at back-up company Acronis - 2012-2013
• Red Hat, Managing Systems Design & Engineering since 2013
Greybeards fight Balrogs. They hate systemd. They fork distributions.
The Role of the Linux OS

Infrastructure or Application Platform?

- In abstract representations of the modern software stack, the OS is often considered part of the Infrastructure.

- However, an alternative, application-centric view would consider it's primary role to provide a common runtime for applications, abstracting from infrastructure.
Breaking the vertical lock-in of Mainframe & Mini-Computers, UNIX

**MAINFRAME**
Complete vertical integration
Vendor-controlled
HW/OS/Ecosystem.

**UNIX**
Vertical integration of infrastructure & app platform
Semi-open ecosystem.

**Linux - e.g. RHEL**
Completely Open HW and ISV ecosystem with the Linux OS as the neutral enterprise app platform
In the beginning there was /usr/local/ - and stow, and binaries mounted on NFS.

- Servers were special pets. - They were dog-show exhibits.
  - Inherited from Unix host tradition.
- Software often compiled on the production machine.
- High-maintenance.
- Fragile due to dependencies on each host's environment:
  - Application behaviour depends on the state of the individual machine.
  - Not efficient for managing artifacts.
- Late-binding based on source-level API.

Doesn't scale in distributed environments (aka PCs).
Scalability Through Binary Packaging

Then, There Be RPM and up2date, yum, dpkg, and apt...

- Frozen binary distribution, reproducible builds.
  - Build once, distribute binary across multiple Linux servers.
  - Metadata, signatures.
  - Predictable behavior, dependency management.
  - Management of installed artifacts, updates.
  - Transport for a curated content stream from a trusted source.
- Implicit lock into single instance, single version monolithic userspace.
- Implements a late-binding model for deploying software in Ops based on an ABI contract.

Welcome to Dependency Hell.
Finally kickstart, satellite, cfengine, and the likes...

- Mass deployment and recipes
- Efficiency through automation. Binary distribution at scale.
- Volatility of late-binding dependency resolution, conflicts & compatibility.
- Automate the stack composition on machines.
- Manage the lifecycle of the software stack.
- Centralize management control.
- Components move across dev/test/ops independently.
- Still in Dependency Hell.

Model still largely used today, sometime with the same components plus newer tools like Ansible.
Virtualization, Appliances, The Cloud - Everything is a VM

- Common model: Deploy as pre-built images, operate as pet.
- Predictable initial stack behaviour.
- Existing tools continue to work - it’s just virtual HW.
- Multiple instances, multi-tenant.
- Still monolithic inside the VM, still dependency conflicts in VM

Less Dependency Hell - Hello VM Sprawl and inconsistent management.
Infrastructure Elasticity

- Efficient sharing of physical HW due to sharing infrastructure.
- Often Linux inherited one VM per service from Windows.
  - Multi-tier applications consisting out of multiple service.
  - Heavyweight compared to running multiple processes in a single instance.
- Efficient cluster management on VM-level, ‘Software Defined’ Datacenter
- Potentially the a single artifact to move across DEV/TEST/PROD if integrated into a full image-based lifecycle.
- Did we say Sprawl, but The Cloud takes the ops problems off your hands.

Move towards service aggregation, vertical integration.
Shifting Paradigms

MACRO TRENDS

- “Software is eating the world”
- Business-value driven developers gaining influence over traditional IT
- Open source is the default; driving rapid growth in content volume and stack complexity

PREFERENCES & BEHAVIOR

- Move towards Cloud Native behaviors
- Aggregation of services replaces monolithic systems
- Preference to consume most current versions
- Shift from a broadcast-model to an on-demand model, SaaS

TECHNIQUES & TOOLS

- DevOps enables developers to manage rapid pace of change
- Containers creates application-centric runtimes that allow maximum flexibility with minimal overhead
Software Stack Complexity Is Growing

Module Counts

Source: www.modulecounts.com
Traditional Distro vs App-Centricity

Diminishing Returns at Growing Complexity

Traditional binary software distribution great for foundational platform components...

But:

- Modern software stacks have become too complex to be mapped into a common, monolithic namespace.
- As a developer, I have to go to native packaging (e.g. npm) anyways because the distribution does only provide a small part of what I need to build my application.
- Slow delivering new versions to app developers.
- The higher in the stack, the bigger the issue.
- Re-packaging, frozen binary distribution offers little value for the App developer.
- Upstream binary/bytecode formats sufficient, they compile their software anyways, lock-in for hybrid environments.
- Testing is more valid if done with the actual application, using it.
Customers are increasingly operating in both environments; we must remain relevant in both

**Ops-Centric Environments**
- Stability through control
- Single-stream and generational-based distribution
- Approval-based governance
- Generational, planned appdev
- Optimized for the traditional hardware-centric, monolithic host deployment model
- **Download to install and update in place**

**Application Environments**
- On-demand distribution
- Stability through validation
- Continuous and process-based governance
- Rapid prototyping and iterative development
- Optimized for cloud-ready applications where infrastructure is ubiquitous
- **Download to build**
Liberation: Containers

Expanding use of containers, from VServer over LXC to OCI

- Separate the application runtimes from system runtime.
  - Like chroot but with an epstein drive.
- Multi-instance, multi-version environment with possible multi-tenancy: each service has its own binary runtime.
- Light-weight - at the end, it’s just Linux processes separated by kernel features: CGroups, Namespaces, SELinux

Good bye Dependency Hell
OCI Containers provide the package format for Application-Centric IT

- Aggregate packaging deployed into containers.
  - Initiated by the project previously known as ‘Docker’.
  - Combine existing Linux Container technology with Tar + overlays -> Unicorns
- Frozen binary distribution, reproducible builds.
  - Build once, distribute binary across multiple Linux servers.
  - Metadata, signatures.
  - Management of installed artifacts, updates.
  - Transport for a curated content from a trusted source.
- Fully predictable stack behaviour, life cycle, lightweight.
- Implements an early-binding model for deploying applications packaged by a developer. CI/CD friendly.
- Now solidified with independent runtime: CRI-O

The best of both worlds.
Multi Container Apps

In reality, most applications consist of multiple containerized services.

- Ideal container is only a single binary.
- Applications are aggregated from multiple containerized services.
- Ideal for cloud native applications. Hybrid model for existing apps.
- From multi-tier applications to micro services.
- Static linking or dynamic orchestration.

Great to solve dependency hell, but how to make sure my frontend knows which database to talk to?
The Cluster Is The Computer

By default, everything is a cluster

- Kubernetes manages containerized services across a cluster of Linux nodes.
- Application definition model, describing the relationship between services in abstraction from the individual node.
  - Abstraction from the underlying infrastructure: compute, network, storage.
  - Same application definition remains valid across changing infrastructure.
- Whole stack artifacts move across dev/test/ops unmodified.
- Scale-out capabilities and HA are commoditized into the standard orchestration.
- Often built around immutable infrastructure models.
Talking of packaging... Now we have to package and deliver multi-container-apps:

- **Single containers are just individual services.**
  - Ideally a single process in a container, although in reality often multi-process containers with embedded init-system.
- **Users want to run applications. That requires...**
  - the container images
  - the Kubernetes orchestration data
  - the configuration and parameterization of the service components.
- **Example:** `dnf -y install freeipa-server && ipa-server-install`
  - Answer a bunch of questions and viola you have an Open Source alternative to AD running. - Now try that with containers - well, not with a single monolithic container....
- **Ansible Playbook Bundles or Helm - Two evolving efforts that I see promising:**
  - [Helm and it’s Charts](https://helm.sh) concepts - lacking the ability to configure services?
  - OpenServiceBroker API for Kubernetes with the [Ansible Service Broker and its Ansible Playbook Bundles](https://github.com/ansible/service-broker). - Bringing together the goodness of Kubernetes and Ansible.
Meet the Kubernetes Service Catalog
# Container Use Cases

## Three Major Types of Container Use Cases

<table>
<thead>
<tr>
<th>FULLY ORCHESTRATED MULTI-CONTAINER APPLICATION</th>
<th>LOOSELY ORCHESTRATED CONTAINER APPLICATION</th>
<th>PET CONTAINER</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Container build service</td>
<td>- Container build service</td>
<td>- Environment bootstrapped by container</td>
</tr>
<tr>
<td>- Multi-container apps</td>
<td>- One or multi-container</td>
<td>- Run YUM, NPM, etc inside container instance (NOT pre-built in build service)</td>
</tr>
<tr>
<td>- Kubernetes (E.g. Red Hat OpenShift)</td>
<td>- No Kubernetes</td>
<td>- Ansible etc. inside container (similar to VM use case)</td>
</tr>
<tr>
<td>- Central management engine (e.g. ManageIQ)</td>
<td>- Static relationships configured on the host</td>
<td></td>
</tr>
<tr>
<td>- Ansible (or Puppet / Chef / Salt) used against higher-level APIs</td>
<td>- Wrappers such as the Atomic tool</td>
<td></td>
</tr>
<tr>
<td>- Packages as APBs or Helm Charts.</td>
<td>- Ansible, etc. used against lower-level APIs</td>
<td></td>
</tr>
</tbody>
</table>

- Environment bootstrapped by container
- Run YUM, NPM, etc inside container instance (NOT pre-built in build service)
- Ansible etc. inside container (similar to VM use case)
The New App-Centric Platform

Built around the Application Platform and Application Content.

Enabled by developer tools and hybrid cloud management.

Fed by packaged service and developer content ecosystems.

Supported by infrastructure integration.
Kubernetes (in the Red Hat case in the form of OpenShift) provides a scale-out, “the cluster is the computer” platform to deploy fully-orchestrated multi-container applications.

Built on the Linux OS (here RHEL Atomic Host) in the immutable infrastructure paradigm, OCI containers, etcd, kubernetes, systemd.

Application is defined in abstraction from Infrastructure provider details, works across different cloud providers, integrates with infrastructure services.

Fully Open Source, Standards-based, pluggable.
Applications consist of multiple containers, built as OCI Container images. Kubernetes defines application entity.

Ansible Application Bundels and the OpenServiceBroker API provide a transport model for full-application portability across the ecosystem.

Userspace portability across different versions of the Host. E.g. Red Hat provides RHEL images to build apps on. Today support for RHEL 6 and 7.

Pre-packaged services can be easily aggregated without need to build images or application templates.
Trusted Ecosystem

OPEN ECOSYSTEMS
Content available from vendors such as Red Hat, partners offering specific services, community. Selection controlled by customer.

COMPONENT-LEVEL
Downloaded to build E.g. Java Library, RPM, zip, pip, npm module, ...
“As a developer, I want to develop my application using an existing UI framework library.”

PACKAGED SERVICES
Downloaded to install OCI Container images and metadata wrapped as APBs from Container Registries in early-binding model..
Examples: Database, messaging service, platform extensions, host drivers.
“As a sysadmin or developer, I want to aggregate a pre-packaged database service into my application.”
Public cloud & proprietary private cloud are driving vertical integration and lock-in with pseudo-standards.
- Just like UNIX.

Kubernetes is in the unique position to again become the neutral runtime for an open ecosystem on hybrid infrastructure, disrupting the vertical integration of proprietary vendors.
- Just like Linux.
And then the Infrastructure...

Kubernetes is expanding into Infrastructure Services

Two examples:

- **Kubevirt** - Community project for VMs orchestrated as containers by Kubernetes.
  - VMs are just special cases of containers.
  - Hybrid, converged infrastructure.

Conclusions

• OCI Containers are now part of the standard Linux experience.
• The default deployment is shifting from single machine to clusters.
• Kubernetes is becoming the default orchestrator for these clusters.
• The traditional approach of Linux distro packaging will change:
  – Multi-instance multi-version on a single system.
  – Native upstream packages (npm, gem, pip, ...) inside containers.
  – Source-to-container build models are evolving.
• Full application portability across hybrid clouds will the key to counter vertical lock-in.
The future of the Linux OS is a scale-out cluster-as-computer platform for fully orchestrated multi-container apps, providing an abstraction layer across underlying infrastructure, and breaking the vertical integration of proprietary cloud.
Thank you!