Pros/Cons of Deploying Data into Containers

Dave Nielsen @davenielsen
Redis Labs @redislabs
Agenda

• The Cloud Perspective
• Why VMs?
• Why Containers?
• Why Data in Containers?
Benefits of Cloud for new apps are undeniable.

- More efficient, faster deployments of servers
- Some solutions are offered serverless (As-A-Service)
- Agile, faster development = better solutions
- Many tools / frameworks leverage current best practices
- Best programmers use current / better tools

“You should expect better ROI. Even if you pay a little more!”
Of course, bleeding edge does not always work out
Too much change, too fast, can cause problems too
Benefits for existing apps are less certain
Might just leave where they are
But for new apps, we’re all on board, right?

Ok. Now what?
If Cloud, then why Virtualization?

Cloud Computing Options (Server Virtualization)

- Bare metal clouds? Not many
- VMs are predominant

VMs

- Run heterogeneous OS with a variety of sizes (CPU/Memory) on the same physical machine = Wide variety of apps on the same servers
- Use same infrastructure, deployment process, security, monitoring, backup/recovery process for many/all applications
- Significant operation improvement
Challenges of VM-based Cloud

Manage many VM images and instances
Worry about security breaches via the OS
• Patches & Upgrades
• Viruses & Worms
Lots of Custom Scripts
• Restarting Services
• Backup & Restore
• Command Line Tools
Still a lot can be automated
Cloud Computing Redux: Containers
About Containers

Containers have been around for a long time (Sun, Parallels). Then Cloud vendors began to use them (Heroku, Google, Joyent). Recently, mature/reliable enough for general use.

- Docker, CoreOS,

Now large enough to attract mass market developer tool vendors.

- Microsoft, VMware, Amazon, Google, IBM, Red Hat, Canonical,
Benefits of Containers

• Still have multiple “containers” ... but fewer # of variations of OS
• Fewer Patches & Upgrades
• Fewer Viruses & Worms
Benefits of Containers

New Benefits

• Small, light weight, easy to share (developers, Q&A, production)
• Dependencies included in container
• Easy to share a container – Big Deal!
• Public register, never had a public before
• Spins of fast. Provision/de-provision processes (<1 secs)
  • Great for Microservices!!
• Runs on every cloud
• Security - Fine grained, smaller attack vector, less exposure
Microservices at Netflix
Microservices: Netflix, Gilt Group, Twitter
Docker Workflow/Sharing: Build Push Pull Run

via Jérôme Petazzoni of Docker
Application Code, Supporting Code, in Unity File System

Linux
Who decides?

- Developer?
- Architect?
- IT?
Buildpacks

- Created & Open Sourced by Heroku
- Catching on (Heroku, Cloud Foundry)
- Provides runtimes (JVMs, etc) for your code
- Dockerfiles/Buildpacks include:
  - Dependencies that need to be downloaded
  - Configuration variables
  - Specify the runtime options
  - Scripts that examine developer-provided artifacts
DevOps Workflow

• Traditional Team w/ Dev & Ops
• Put buildpacks in containers / Or use Dockerfiles
• Develop code in containers
• QA code in containers
• Deploy Production code in containers
# The New App Stack

<table>
<thead>
<tr>
<th>Application Code</th>
<th>Buildpack in Container</th>
<th>Linux</th>
</tr>
</thead>
</table>

Microservices Workflow

- Team owns Dev & Ops
- Put everything in containers
- Build Container, then push through workflow
Microservices App Stack

Application Code & Buildpack in Container

Linux
Docker Workflow: Build Push Pull Run

- **Build**
  - Source Code Repository
  - Dockerfile For A
- **Push**
  - Container A
  - Docker Registry
- **Search**
  - Pull
  - Prod Hosts: OS (Linux)
- **Run**
  - Container A
  - Container B
  - Container C
Q4 Which container technology have you used or investigated?
Answered: 254   Skipped: 31

What about Multiple Containers

Many types of Developers and skill levels

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
<th>PHP</th>
<th>Ruby</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Java image" /></td>
<td><img src="image2" alt="C image" /></td>
<td><img src="image3" alt="PHP image" /></td>
<td><img src="image4" alt="Ruby image" /></td>
</tr>
<tr>
<td><img src="image5" alt="Java image" /></td>
<td><img src="image6" alt="C image" /></td>
<td><img src="image7" alt="PHP image" /></td>
<td><img src="image8" alt="Ruby image" /></td>
</tr>
<tr>
<td><img src="image9" alt="Java image" /></td>
<td><img src="image10" alt="C image" /></td>
<td><img src="image11" alt="PHP image" /></td>
<td><img src="image12" alt="Ruby image" /></td>
</tr>
<tr>
<td><img src="image13" alt="Java image" /></td>
<td><img src="image14" alt="C image" /></td>
<td><img src="image15" alt="PHP image" /></td>
<td><img src="image16" alt="Ruby image" /></td>
</tr>
</tbody>
</table>

as seen by...

- Java fanboys
- C fanboys
- PHP fanboys
- Ruby fanboys
Developers are asked to do Dev & Ops:

```bash
@Doni@/.ec2$ ec2-stop-instances i-ee9bed61 -f
INSTANCES i-ee9bed61 stopping
@Doni@/.ec2$ ec2-describe-instances i-ee9bed61
RESERVATION r-081797ed 243323232012 default
INSTANCES i-ee9bed61 ami-08728661 ec2-67-202.11.20
e-1.internal stopping devda 0
07a9529 monitoring-disabled 67.202.12.11
aravirtual xen sg-e9f99a1 default
BLOCKDEVICE /dev/sda1 vol-a8d6ec2 2011-04-15
BLOCKDEVICE /dev/sda vol-dbdc30b0 2011-04-15
TAG instance i-ee9bed61 Name
@Doni@/.ec2$
```
Which is ok for some
But difficult for most
And ... there’s a new generation of developers ...

... and they want something easy ...
... and PaaS is the easy button.
... which might not always work out ...

... the way you expect.
Container Orchestration Engine & Platforms

- Fewer unique OS to worry about
- Services are provided by the COE or PaaS
  - Don’t’ have to setup database, storage
- Redundancy, backup, etc. is built in
- Allocating IP addresses, sub-domain, etc.
- IDE Integration with PaaS Platforms
Available Container Orchestration Engines

**Kubernetes** (by Google). Also multiple implementations, OpenShift, Deis, etc.
- Enhanced network - easy to connect containers across hosts
- Opinionated - enforces several concepts

**Swarm** (by Docker):
- Native clustering for docker - supports docker APIs
- Still has issues with connecting dockers across hosts and with scheduling tasks

**Mesos** and **Marathon** (by Mesosphere):
- Mesos is a low-level, battle-hardened scheduler that supports several frameworks for container orchestration including Marathon, Kubernetes, and Swarm
- Used for managing large-scale docker/containers environments (Twitter, eBay, Airbnb)

**Fleet** (by CoreOS)
- Low-level and fairly simple orchestration layer that can be used as a base for running higher level orchestration tools, such as Kubernetes or custom systems

**Cloud Foundry** (Cloud Foundry Foundation); Pivotal, IBM, HP, SAP, GE, etc.

Benefits of Container Orchestration Engine

- Standardized cluster deployment / cluster operation rather than standardized instances deployment (like Chef and Puppet)
- Makes sure resources are efficiently used (not sitting idle), but can still cope with on-demand spikes
- Handles resources rebalancing, scheduling, scaling & replication
- Supports various affinity and constraint rules
I’m not a security expert
Summary

Containers vs VMs
• Deploying standardized containers onto a container framework. Gives a standardized environment which you can use without thinking about.

Data in Containers
• Many options. Persistent storage is important difference.
• Separate deployment is likely the best
Why not to deploy data via containers

1. In many cases Database needs to be deployed on high-memory instances
   → not required by most Apps
2. When Database data-persistence is enabled, these instances should also be
   connected to external storage devices, like AWS EBS or SAN with high IOPS
   capabilities (e.g. 10K IOPS)
   → unused for most Apps
3. Databases are a heavy consumer of network resources - in a typical production
   environment Database container should run on an instance that supports ~1M
   packets/sec
   → overkill for most Apps
4. Mixing Database and App containers on the same HW/VM might lead to:
   a. Processing bottleneck - if both containers are deployed on the same unsuitable node
   b. Expensive infrastructure - if each node in the deployment can potentially host a Redis container,
      due to the constraints mentioned in [1],[2] and [3] above
Example of a Inefficient Deployment of Database with Kubernetes

Deploying Database and App containers on the same node is a bad practice
A Better Option for Deploying Database with Kubernetes

But still with many issues

(1) not fully HA; (2) cannot scale; (3) scheduling is not based on Database specific metrics and insight; (4) lack of major database specific cluster management processes: migration, resharding and balancing based on redis status/usage/metrics; (5) limited number of database containers per node
Optimal Database Deployment on Containers – Kubernetes, etc.

App on Kubernetes Cluster

RLEC on Kubernetes Cluster

Services

Kubernetes Pod 1
- app container 1
- app container 2

Kubernetes Pod 2
- app container 1
- app container 2

Proxies Pod

CMs Pod

Redis shards, each has its own Pod
THANK YOU / QUESTIONS

Dave Nielsen @davenielsen
Redis Labs @redislabs
Additional Resources

http://www.cloudtp.com/2015/11/30/docker-container-landscape/