Advanced Continuous Delivery Strategies for Containerized Applications Using DC/OS

ContainerCon
@ Open Source Summit
North America 2017

Elizabeth K. Joseph
@pleia2
Elizabeth K. Joseph, Developer Advocate

- Developer Advocate at Mesosphere
- Spent 4 years working on CI/CD for OpenStack
- 10+ years in Linux systems administration and engineering roles
- Author of The Official Ubuntu Book and Common OpenStack Deployments
Continuous delivery (CD) is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time.

Via https://en.wikipedia.org/wiki/Continuous_delivery
Modern Release Process

Better products through a repeatable release cadence

Happier developers through continuous feedback
CD: A Key Component of Modern Release Processes

Continuous Delivery - getting workloads READY and RUNNING:

- Perform code analysis, unit tests, and integration tests (continuous integration)
- Dynamically provision environments, configure them, and manage dependencies
- Provision servers (infrastructure automation)
- Deliver and Deploy applications to environments (Dev to Stage to Prod)

- Low risk releases
- Faster time to market
- Higher quality SW
- Lower costs
- Happier teams
CD with Containers and DC/OS: 2-pronged approach
Run everything in containers!

CC BY 2.0: https://www.flickr.com/photos/rubbermaid/6909787969/
Organize everything efficiently!
**Traditional Workload Flow Stages**

<table>
<thead>
<tr>
<th>Developers</th>
<th>Operator Managed (Shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional Monoliths</td>
<td></td>
</tr>
<tr>
<td>Install (Local)</td>
<td>Test CI</td>
</tr>
<tr>
<td>Development (Local Deploy)</td>
<td>Stage (pre-prod) CD</td>
</tr>
<tr>
<td></td>
<td>Production CD</td>
</tr>
</tbody>
</table>
## Modern Workload Flow Stages

<table>
<thead>
<tr>
<th>Developers (Local, Shared)</th>
<th>Operator Managed (Shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stateless</strong></td>
<td></td>
</tr>
<tr>
<td>Install (Local)</td>
<td>Stage (pre-prod)</td>
</tr>
<tr>
<td>Development (Local Deploy)</td>
<td>Test</td>
</tr>
<tr>
<td>Test</td>
<td>Production</td>
</tr>
<tr>
<td><strong>Stateful DB</strong></td>
<td></td>
</tr>
<tr>
<td>Install (Local)</td>
<td>Stage (pre-prod)</td>
</tr>
<tr>
<td>Development (Local Deploy)</td>
<td>Test</td>
</tr>
<tr>
<td>Test</td>
<td>Production</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
</tr>
<tr>
<td>Install (Local)</td>
<td>Stage (pre-prod)</td>
</tr>
<tr>
<td>Development (Local Deploy)</td>
<td>Test</td>
</tr>
<tr>
<td>Test</td>
<td>Production</td>
</tr>
</tbody>
</table>
BUILDING & OPERATING CI/CD PIPELINES IS CHALLENGING

- Installing each service and maintaining upgrades is time-consuming, with each machine having different OS’s and tooling
  - More difficult because teams like to use many technologies and tools as building blocks
  - Spinning up CD pipeline for each application is time-consuming
- Low utilization driven by silos of developers with single-instances of tools
- Poor allocation of capacity may prevent developers from shipping code, and acquiring new HW is slow
NAIVE APPROACH

Industry Average
12-15% utilization

Typical Datacenter
siloed, over-provisioned servers, low utilization
THE KERNEL: APACHE MESOS
Use: The primary resource manager and negotiator

Why Mesos?

- 2-level scheduling
- Fault-tolerant, battle-tested
- Scalable to 10,000+ nodes
- Created by Mesosphere founder @ UC Berkeley; used in production by 100+ web-scale companies

DC/OS: Datacenter Operating System

- Resource management
- Task scheduling
- Container orchestration
- Logging and metrics
- Network management
- “Universe” catalog of pre-configured apps (including Jenkins, GitLab, Artifactory…), browse at [http://universe.dcos.io/](http://universe.dcos.io/)
- And much more [https://dcos.io/](https://dcos.io/)
DC/OS Architecture Overview

Services & Containers

- HDFS
- Jenkins
- Marathon
- Cassandra
- Flink
- Spark
- Docker
- Kafka
- MongoDB
- +30 more...

DC/OS

- Container Orchestration
- Security & Governance
- Monitoring & Operations
- User Interface & Command Line

ANY INFRASTRUCTURE

- Physical Servers
- Virtual Servers
- Private Cloud
- Public Cloud Providers (Google, AWS, Azure)
Interact with DC/OS (1/2)

Web-based GUI

https://dcos.io/docs/latest/usage/webinterface/
## Interact with DC/OS (2/2)

<table>
<thead>
<tr>
<th>CLI tool</th>
<th>API</th>
</tr>
</thead>
</table>
MULTIPLEXING OF DATA, SERVICES, USERS, ENVIRONMENTS

Typical Datacenter
- siloed, over-provisioned servers,
- low utilization

Mesos/ DC/OS
- automated schedulers, workload multiplexing onto the same machines
Development Team Self-Service for CI/CD

RELIABLE, SIMPLIFIED CI/CD INTEGRATION with DC/OS

Continuous Delivery Pipeline

Version Control System → Continuous Integration → Container Orchestrator → Production Environment → Load Balancer

Artifact Repo & Container Registry

Apache Mesos & DC/OS

git push
Development Team Self-Service for CI/CD

RELIABLE, SIMPLIFIED CI/CD INTEGRATION with DC/OS

Continuous Delivery Pipeline

1. GitLab, Bitbucket, GitHub
2. Jenkins
3. Marathon
4. DC/OS (Mesos)
5. Marathon-lb

Apache Mesos & DC/OS

git push
Development Team Self-Service for CI/CD
LET DEVELOPERS USE THE TOOLS THEY WANT

- Single-command installation of services like Jenkins, GitLab, and Artifactory
- Once a service is installed, it can be run across the entire datacenter, elastically sharing all or some of the datacenter’s resources
- Ability to run application code (PaaS), containers, and distributed applications with no restrictions to application development teams
A MODERN RELEASE PROCESS

1. Development Team Self-Service for CI/CD
   - Scale services instances to provide on-demand Build/Test/Staging infrastructure with reduced time & cost to provision
   - Manage multiple installations for different groups; centralized role based access control to all cluster resources
   - Choose the tools you want and get support from partners for enterprise tools integrated with DC/OS

2. Elastic Scaling with Resource Optimization for build bursting
   - Teams share the same pool of resources, dramatically increasing utilization (6,000 builds/day on 46 physical machines - eBay)
   - Use CI/CD tools of your choice with DC/OS, and run everything on the same shared infrastructure
   - Health checks to ensure developer tools are always up and running; if an instance fails, it is automatically restarted without data loss

3. Build and deploy traditional and modern apps on the same infrastructure
   - Identical infrastructure across Test/Staging/Production with strong isolation
APPLICATION LIFECYCLE

Build and deploy modern apps on the same infrastructure.

**DC/OS**
- Identical Infrastructure across Test/Staging/Production with strong isolation
- Self service access

**BENEFITS**
- Less developer time troubleshooting environment issues
- Easy experimentation with new technologies
## Deploying Apps

### Manual

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Deployment</th>
<th>Health checks</th>
<th>Service discovery</th>
<th>Persistence</th>
</tr>
</thead>
</table>
| ● A sysadmin provisions one or more physical/virtual servers to host the app | ● By hand or using Puppet / Chef / Ansible  
● Jenkins SSHing to the machine and running a shell script  
● Note: all dependencies must also be present! | ● Nagios pages a sysadmin | ● Static hostnames / IP addresses in a spreadsheet or config management  
● A sysadmin configures a load balancer manually or with Puppet / Chef / Ansible | ● Individual servers with RAID 1/5/6/10, expensive SANs, NFS, etc.  
● Dedicated, statically partitioned Ceph or Gluster storage clusters |

### Automatic

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| ● Mesos resource offers (two-tier scheduling) offers available resources directly to frameworks | ● Containers deployed, ideally using a CI/CD tool to create/update app definitions | ● Health checks, restarts unhealthy/failed instances | ● Provides DNS resolution for running services (hostname / IP address, ports, etc)  
● Load balancer configs built dynamically using cluster state | ● External/persistent volumes (REX-Ray), HDFS, etc.  
● Self-healing Ceph or Gluster on Mesos / DC/OS |
**Old vs. New Deploy Process**

<table>
<thead>
<tr>
<th>Pre-Container</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build a Debian package</td>
<td>Build and push Docker image</td>
</tr>
<tr>
<td>Push deb pkg to apt server</td>
<td>Deploy</td>
</tr>
<tr>
<td>Wait for apt server to have deb package ready</td>
<td>&lt;1 minute</td>
</tr>
<tr>
<td>Boot a new AWS instance</td>
<td></td>
</tr>
<tr>
<td>Run puppet on the instance, installing the deb</td>
<td></td>
</tr>
<tr>
<td>Turn the instance into an AMI image</td>
<td></td>
</tr>
<tr>
<td>Boot new AWS instances using new AMI</td>
<td></td>
</tr>
<tr>
<td>Terminate old AWS instances</td>
<td></td>
</tr>
</tbody>
</table>

“*It would easily take 30 minutes for a single deploy even under ideal conditions where nothing broke.*”

“*A simple service might only take 20 seconds to fully deploy under ideal conditions.*”
Questions?

Elizabeth K. Joseph
Twitter: @pleia2
Email:
ejoseph@mesosphere.com

@dcos
chat.dcos.io
users@dcos.io
/dcoss
/dcoss/examples
/dcoss/demos