An Automation Development Stack

One implementation of an automation and configuration stack that allows users to leverage a complex stack via a high level of abstraction.

Christopher Steel, MCIN ADM, | December 14th, 2018 | src | pdf |

Obtaining A High Level Of Abstraction

In this case, obtaining a high level of abstraction:

- Makes it "easy" to leverage the magical powers of some complex technology
- Allows for the creation of provider agnostic automation scripts which in turn
- Leads to standardized automation roles
- Likely to increase velocity

Not exactly magic but...

"Any sufficiently advanced technology is indistinguishable from magic."

Arthur C. Clarke's Third Law

- No rabbits, just a software stack and
- A few virtual machines and containers and
- Perhaps a COW or two?!

Development Stack Overview

- Ansible - automation software
- LXC / LXD - container system and hypervisor
- Molecule - Ansible role development aid
- OpenZFS - Powerful file system and logical volume manager
- VirtualBox - Opensource Virtual Machine application

Ansible

An open source automation engine

- Easy to grok
- Provisioning and configuration management
- Application deployment and intra-service orchestration
- Modules in abundance made using any language
- SSH / No agents required

LXC and LXD

Linux System Containers

- As fast as bare metal (2% slower).
- Ultralight hypervisor
- Well considered CLI / REST API.

Molecule

- Allows for consistently developed Ansible roles
- Supports multiple:
  - instances
  - operating systems
  - distributions
  - test frameworks
  - testing scenarios
  - virtualization providers / target nodes (bare-metal, VMs, cloud(s) and/or containers)

OpenZFS

OpenZFS acts as both a volume manager and a file system it has a few magical properties as well.

- Copy-on-write (COW) transactional object model
- Continuous integrity checking and automatic repair
- Lots of other interesting stuff, RAID-Z, Native NFSv4 ACLs...
- Ubuntu has a really nice OpenZFS, FreeNAS an appliance version.

VirtualBox

- Open source
- Well understood
- Mature
- Runs on many OS, able to host many OS...

Virtualization Overview

Before we get started with the details of this particular development stack, lets take some time and talk a bit about virtualization and how to go about choosing the right type of virtualization for your particular project or experiment.

Reasons you want (need) to virtualize?

- Isolation / security
- Density / effective use of resources
- Efficiency and/or speed
- Repeatability and/or portability
- Required in order to...
- Other reasons

Hypervisors

What is a Hypervisor? Generally speaking a Hypervisor:

- Allows a physical host to operate multiple VMs or containers as guests
- Is a process that separates a system's OS and apps from the physical hardware
- Is software, but could be embedded software on a chip

Major types of Virtualization (Hypervisors)

- Type 1 - Native (bare metal)
- Type 2 - Hosted ("on top of" OS)
- Type C - OS level virtualization / Containerization / Other

Type 1 - Native ("Bare Metal")

```
Hardware <--> Hypervisor <--> Hosted OS
```

- Runs directly on the hosts hardware
- Controls the hardware
- Manages the guest operating system(s)

Type 1 (Bare Metal) Hypervisor Examples:

- VMware ESXi
- Xen (open source - loads it's own paravirtualized host operating system)
- Xbox One system software

Type 2 - Hosted Hypervisors

```
Hardware <--> Host OS <--> Hypervisor <--> Hosted OS
```

- Generally an application that runs on a conventional (host) operating system OS
- The guest operating system runs as a process on the host OS
- Abstracts the guest OS from the host OS

Type 2 (Hosted) Hypervisor Examples:

- KVM
- VirtualBox
- VMware Workstation Pro

Some properties of Type 2 Virtualization

- Isolation of instances
- Support for non-linux OS
- Well understood, familiar to more users

Type C Virtualization

Catch all including: containers, linux system containers, jails and chroot jails, AIX Workload Partitions (WPARs) and virtual environments.

- Diverse but shared history
- May make use of a hypervisor
- May allow for the existence of multiple isolated user-space instances

Containers, Type C architectures that employ a hypervisor

Instances (launched containers) look like real computers from the point of view of the programs running in them.:

```
                 _ Hypervisor*

              /          |
HW -- OS -<              |

              \ ____ Hosted Instance (OS / App / Service...)
```

Examples of Type C Virtualization using Hypervisors:

- Docker - Used to be an alternative Hypervisor for using Linux System Containers (LXC)
- LXC (Linux System Containers) - LXD is an extension of the LXC hypervisor

Containers

Selecting the right container for the job

- Usage (microservice, enterprise application, development, devops, HPC?)
- Users (are you a developer, admin, power user or a researcher?)
- Support & Community, Popularity
- Strengths & Weaknesses
- Sharing containers - Image/Container Registry

Singularity

Designed for HPC

- Can be "run" or executed directly by file name
- Image-based containers
- Must be root outside of container to become root inside of container
- No root daemon owned processes
- Can import docker containers
- Learning curve

Singularity Users

- MCIN users
- Calcul Quebec / Calcul Canada
- HPC all over the world

Singularity Pipeline Example

After (a lot of) careful preparation...:

```
singularity run --app cat catdog.simg
```

Output:

```
 Meow , this is Cat
```

Docker

- Designed for isolating applications, microservices...
- Docker to HPC via Singularity
- Use of layers and disabling of persistence results in lower disk IO
- Issues using apps that expect cron, ssh, daemons, logging and other system stuff
- Can use copy-on-write (CoW) for images and containers

Docker Users

- MCIN
- Development and test organizations
- Many, many others

LXC & LXD (Linux System Containers)

- Acts like an OS environment: file systems, init.d...
- Parallelism possible
- Run one or more multi-process applications
- Linux-native, runs well with ZFS.
- Windows (10 with Linux subsystem enabled) and OSX clients

Who uses LXC / LXD

- IT Operators / DevOps
- MCIN
- Medical University of Gdańsk (Maciej Delmanowski) - DebOps project
- Most Canonical Websites

LXC/LXD usage

- We cover this in the next section

Choosing your virtualization target(s)

- Do you have / require root access on your virtualization project?
- What will you be virtualizing? An application, many applications, part of a pipeline?
- Host - Where will your container be running?
- Which OSs will be running on your host and guest systems?

Molecule

Enables the development of provider agnostic Ansible playbooks and roles.

- Top of our stack
- Built-in provider specific Ansible tasks (using Ansible modules)
- Creation of custom providers

molecule init

- **molecule init role** is used to create a new provider agnostic Ansible role and a default scenario
- **molecule init scenario** is used to create additional scenarios for an existing Ansible role

```
molecule init role -r myrole
cd myrole
rm -R molecule/default
molecule init scenario -s default -d vagrant -r myrole
molecule init scenario -s lxd -d lxd -r myrole
```

Generated files

Generating a **scenario** creates a scenario directory and contents. The INSTALL.rst file includes information on any additional requirements molecule has for supporting a particular scenario:

```
ls -al molecule/default/
-rw-r--r-- 1 cjs cjs  260 Dec  5 20:08 INSTALL.rst
-rw-r--r-- 1 cjs cjs  303 Dec  5 20:23 molecule.yml
-rw-r--r-- 1 cjs cjs   64 Dec  5 20:08 playbook.yml
-rw-r--r-- 1 cjs cjs  235 Dec  5 20:08 prepare.yml
drwxr-xr-x 2 cjs cjs 4096 Dec  5 20:08 tests
```

molecule.yml

- Each scenario has a **molecule.yml** file
- By default, a single instance is created called instance
- You may want to call it something more meaningful

Lets have a peek at our `molecule.yml` files

```
cat molecule/default/molecule.yml | more
cat molecule/lxd/molecule.yml | more
```

vagrant default scenario `molecule/default/molecule.yml` customisation example

```
...
platforms:
  - name: myrole-xenial
    box: ubuntu/xenial64
    groups:
      - linux
    cpus: 2
    memory: 4096
    instance_raw_config_args:
      - "vm.network 'forwarded_port', guest: 80, host: 1080"
      - "vm.network 'forwarded_port', guest: 443, host: 1443"
      - "vm.network 'private_network', ip: '192.168.33.10'"
provisioner:
  name: ansible
  lint:
    name: ansible-lint
```

lxd scenario `molecule/lxd/molecule.yml` customisation example

```yaml
...
platforms:
  - name: myrole-xenial
    source:
      alias: ubuntu/xenial/amd64
    profiles:
      - default
    force_stop: false
...
```

`molecule check` **and** `molecule create`

I prefer LXC/LXD scenarios for testing as it works no matter what I am deploying and leverages the superpowers of ZFS.

```
molecule check -s lxd       # do a dry run
time molecule create -s lxd # create any lxd scenario instance(s)
```

Output:

```
    PLAY RECAP *****************************************************
    myrole-xenial              : ok=1    changed=0    unreachable

real    1m23.251s
user    0m13.296s
sys    0m3.572s
```

Other molecule commands

Currently Molecule has a total of 16 high-level commands. Here are three more:

```
time molecule converge -s lxd
time molecule test -s lxd
time molecule lint -s lxd
```

molecule list

Lists status of instances.

```
molecule list
```

molecule destroy

Use the provisioner to destroy the instances.

```
molecule destroy -s lxd
```

Copy On Write Demo

Lets take a look at some of the things taking place "under" molecule. In particular here we examine storage usages and COW.

ZFS Storage pool

Check our space usage

```
lxc storage list
```

Our image file

```
+---------+-------------+--------+--------------------------------
|  NAME   | DESCRIPTION | DRIVER |             SOURCE
+---------+-------------+--------+--------------------------------
| default |             | zfs    | /var/lib/lxd/disks/default.img
+---------+-------------+--------+--------------------------------
```

Space usage

```
lxc storage info default
```

Output example:

```
info:
  description: ""
  driver: zfs
  name: default
  space used: 505.19MB
  total space: 75.95GB
used by:
  containers:
  - c1
  images:
  - d299226f322c9c743bf07a0bfea02a2a1ca018e04350fb9270e94668d1d42
  - ea1d9641ca09f8d7b55548447493ed808113322401861ab1e09d1017e07d4
  profiles:
  - default
```

Launching LXC Containers With LXD

Starting a container called "c1"

```
lxc launch ubuntu:16.04 c1
```

Check our space usage now:

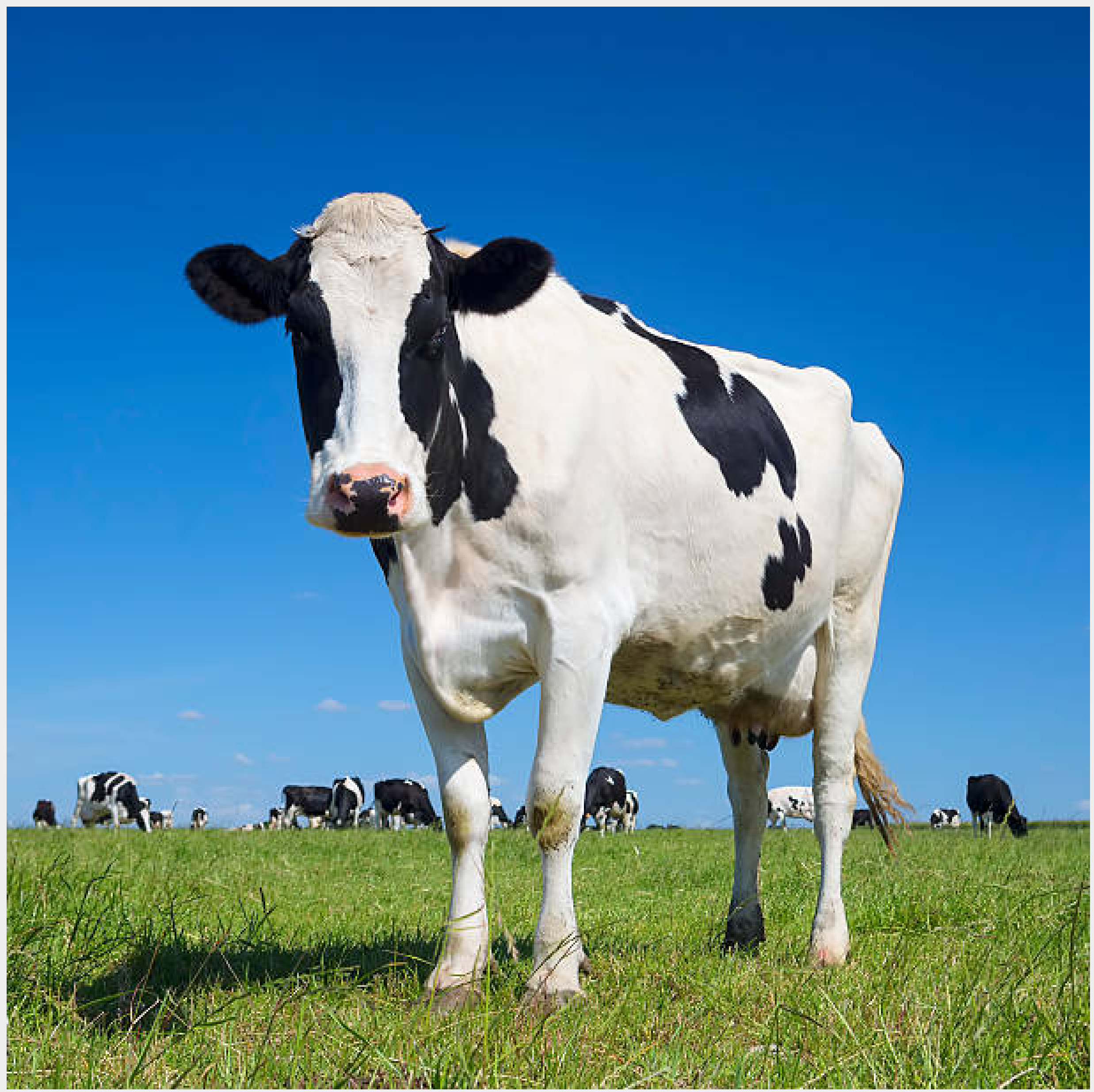
```
lxc storage info lxd
```

List our containers

```
lxc list
```

Some Key Points

- High levels of abstraction rock
- Multiple scenarios makes migrating to and from providers easy
- Allows us to develop provider agnostic Ansible roles
- Increase in velocity



References

Web pages

- https://robin.io/blog/linux-containers-comparison-lxc-docker/
- https://robin.io/blog/containers-deep-dive-lxc-vs-docker-comparison/
- https://www.xenproject.org/users/virtualization.ht

Papers

- Formal Requirements for Virtualizable Third Generation Architectures-10.1.1.141.4815
- Analysis of Virtualization Technologies for High Performance Computing Environments
- Performance Evaluation of Container-based Virtualization for High Performance Computing Environments
- NIST.SP.800-125A-F - Security Recommendations for Hypervisor Deployment
- VIRTUALIZATION TECHNIQUES & TECHNOLOGIES: STATE-OF-THE-ART

Presentations

MCIN

- http://natacha-beck.github.io/cbrain_docker/#/
- https://ibis.loris.ca/Presentations/docker.html#/

Other

- Biondi1-hypervisors.pdf
- What place for the containers in the HPC world ?
- hpc-containers-singularity-advanced
- Live Migration of Linux Containers
- Containers for Science Reproducibility and Mobility-hpc-containers-singularity-advanced
- Streamlining HPC Workloads with Containers
- Type C Hypervisors

```
```