



# Automate Everything - DevSecOps for OS Images with Ansible

## OpenShift Virtualization Image Pipeline

---

Mihai Criveti, CTO @ IBM Cloud Native and Red Hat Solutions, RHCA

October 7, 2021



@CrivetiMihai



## Mihai Criveti, CTO Cloud Native and Red Hat Solutions at IBM

- Cloud Native & Red Hat Solutions Leader
- Red Hat Certified Architect
- Builds multi-cloud environments for large customers

## Base OS Image Automation journey

- Started building OS images using automation in 2010, as Architect for IBM's first Red Hat based cloud.
- Used a horrible combination of Perl and Jenkins for image build process.
- Had to develop base OS images, as well as various middleware images.
- Later on, worked on VMware image builds and had to re-write the pipeline.
- Found the combination of Ansible, Tower and Packer to be an amazing combination to build image workflow.

## Example Workflow: Build, Secure and Test Images for Multiple Environments

### Example automated, layered image build workflow

1. Store all image build scripts, code and artifacts in Git.
2. Code changes or security updates trigger Tower or Jenkins to start a new image build.
3. Ansible will provision and setup a bare metal build node.
4. Parallel builds generate *base* OS images (RHEL 7, 8, Windows, Fedora, etc) when needed (Packer) - for KVM, VMware, VirtualBox various cloud providers.
5. Secondary builds based on the *base* image now trigger (OpenSCAP profiles).
6. Tertiary builds now trigger to install a variety of middleware.
7. Images are packed, signed and uploaded to an image store.
8. Images are provisioned and tested.
9. Dismantle the build infrastructure.

## **1) OpenShift Virtualization - OpenShift managed KVM**

---

# What is OpenShift Virtualization

## OpenShift Virtualization

- A feature of the OpenShift container platform, based on the open-source KubeVirt project.
- Manage Virtual Machines into containerized workloads.
- Develop, manage and deploy VMs side-by-side with containers and serverless on one platform: OpenShift.

## What is KubeVirt?

- Open source project that makes it possible to run virtual machines in a Kubernetes-managed container platform.
- Delivers container-native virtualization by leveraging KVM, the Linux Kernel hypervisor, within a Kubernetes container.
- Provides services like those associated with traditional virtualization platforms, providing the best of both mature virtualization management technology and Kubernetes container orchestration.

# How are OpenShift Virtualization images built

## What images does OpenShift Virtualization support?

- OpenShift images are the same kind of images used by KVM / qemu / qcow2 - and are build in a similar way to OpenStack or RHV images.
- Can also import RHV / VMware images, will use `qemu-img` convert behind the scenes to convert them to RAW and prepare them for OpenShift Virtualization.

## Image Format

- Only RAW and QCOW2 formats are supported disk types for the container image registry. QCOW2 is recommended for reduced image size.
- `virt-sparsify` to help 'compress' the image (sparsify).
- Images can be compressed with `xz` or `gzip`.

## Additional image requirements

- The **QEMU guest agent** is a daemon that runs on the virtual machine and passes information to the host about the virtual machine, users, file systems, and secondary networks.
- **VirtIO drivers** are paravirtualized device drivers required for Microsoft Windows virtual machines to run in OpenShift Virtualization.
- **cloud-init** lets OpenShift customize the image after provisioning - such as setting the Hostname, authorized SSH keys or running a custom script.
- Security patches and your configuration.

# Install OpenShift Virtualization using an Operator

## Install the OpenShift Virtualization Operator and create a cluster

OpenShift Virtualization can be installed on any OpenShift cluster, using the Operator Hub.

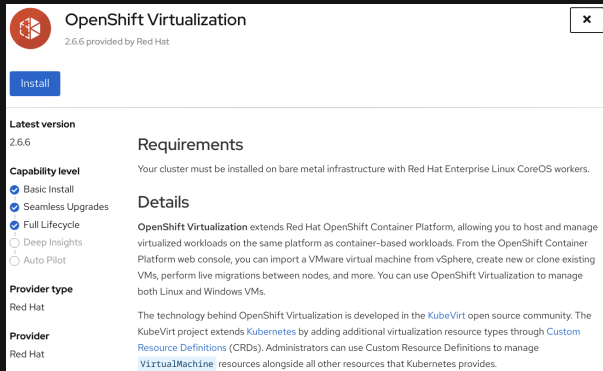
- Install the OpenShift Virtualization Operator (2.4 or higher)
- Create a OpenShift Virtualization Operator Deployment: `kubevirt-hyperconverged cluster` on the Bare Metal node(s).

## Create a new project namespace

```
oc new-project virtual-machines
```

You can now create Virtual Machines by clicking on Workloads > Virtualization.

## OpenShift Operator Hub



The screenshot shows the OpenShift Operator Hub interface for the 'OpenShift Virtualization' operator. At the top, there's a header with the operator's logo, name, and version (2.6.6 provided by Red Hat). Below this is a blue 'Install' button. The main content area is divided into sections: 'Latest version' (2.6.6), 'Capability level' (listing 'Basic Install', 'Seamless Upgrades', 'Full Lifecycle', 'Deep Insights', and 'Auto Pilot' with checkboxes), 'Provider type' (Red Hat), and 'Provider' (Red Hat). To the right of these sections are 'Requirements' and 'Details'. The 'Requirements' section states that the cluster must be installed on bare metal infrastructure with Red Hat Enterprise Linux CoreOS workers. The 'Details' section explains that OpenShift Virtualization extends Red Hat OpenShift Container Platform, allowing for virtualized workloads on the same platform as container-based workloads. It mentions that the technology is developed in the KubeVirt open source community and that administrators can use Custom Resource Definitions (CRDs) to manage VirtualMachine resources alongside other Kubernetes resources.

**Figure 1:** OpenShift Virtualization Operator



## Ephemeral

- When using a `ephemeral` storage volume type, or `containerDisk`.
- The ephemeral image is created when the virtual machine starts and stores all writes locally. The ephemeral image is discarded when the virtual machine is stopped, restarted, or deleted. The backing volume (PVC) is not mutated in any way.

## Persistent (`persistentVolumeClaim`)

- When using a `persistentVolumeClaim`
- Attaches an available PV to a virtual machine. Attaching a PV allows for the virtual machine data to persist between sessions.
- Importing an existing virtual machine disk into a PVC by using CDI and attaching the PVC to a virtual machine instance is the recommended method for importing existing virtual machines into OpenShift Container Platform.

**Note** CDI: Containerized Data Importer.

# CDI: Containerized Data Importer Overview

## CDI Function

- persistent storage management add-on for Kubernetes. Uses `qemu-img` to manipulate disk images.
- provides a declarative way to build Virtual Machine Disks on PVCs for Kubevirt VMs.
- provides a way to populate PVCs with VM images or other data upon creation.
- data can come from different sources: a URL, a container registry, another PVC (clone), or an upload from a client.
- CDI will automatically decompress and convert the file from `qcow2` to raw format if needed. It will also resize the disk to use all available space.

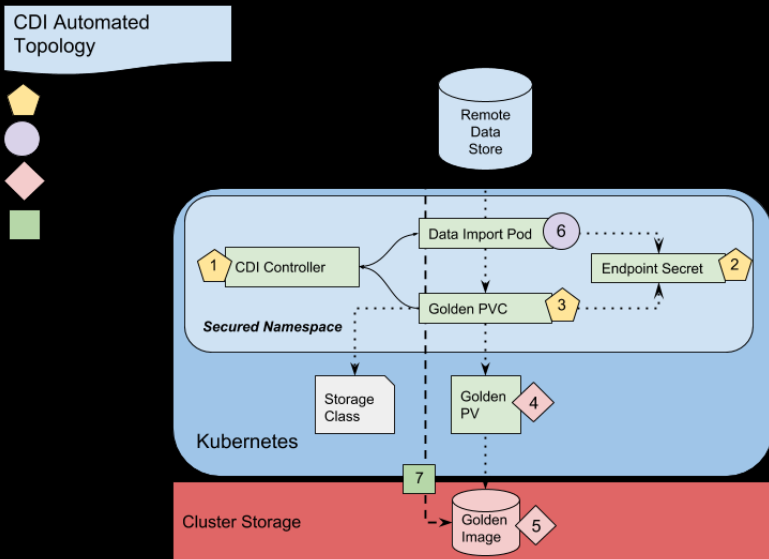
## Import from URL

This method is selected when you create a `DataVolume` with an `http` source. Supports basic authentication (secret) and custom TLS certificates (`ConfigMap`).

## Import from container registry

When a `DataVolume` has a registry source CDI will populate the volume with a Container Disk downloaded from the given image URL.

# CDI: Containerized Data Importer Diagram



**2) Build KVM images by hand -  
what are we automating?**

---

# Architectural Decisions and Image Build considerations

## AD001: Build from scratch (kickstart / sysprep) every time?

- Do we want to rebuild the golden image every time, or do we create a major release build?
- Post-install steps are based on existing golden image (major release).
- Ansible used for post-install steps and parametrization (update the image, secure it, perform various post-install steps).
- Additional 'layers' for middleware or product installs.

## AD002: Clean builds or image conversion?

- When building multi-platform images (ex: VMware, KVM, OpenStack) - do we convert images, or use the right hypervisor for every build?

## AD003: What goes into post-provisioning scripts (ex: cloud-init)

- How much do we want to delegate to cloud-init?
- For example, do we want to bake in the security, patches and compliance in the golden image, or trigger it post provisioning?
- Triggering it after provisioning can be slow, consume bandwidth and expose the image during the update process.

# Setting up a bare metal build host

## Provision a BM host and install KVM, qemu-img and various build tools

- Consider the number of builds running in parallel when sizing the host.
- Bare Metal is recommended for performance considerations.
- Consider using solid state storage for the build host.

```
sudo yum install @virt virt-top libguestfs-tools \
    virt-manager virt-install virt-viewer qemu-img
sudo systemctl enable --now libvirtd
```

## Install virtctl client on RHEL 7

```
sudo subscription-manager repos --enable rhel-7-server-cnv-2.4-rpms
sudo yum -y install kubevirt-virtctl
```

## Install the virtctl client on RHEL 8

```
sudo subscription-manager repos --enable cnv-2.4-for-rhel-8-x86_64-rpms
sudo dnf -y install kubevirt-virtctl
```

# Building a Red Hat OS Image with Kickstart

## Create a Kickstart file (response file)

- You can manually install the OS to generate a .ks file.
- Installations from kickstart are automated, and you could use this as part of a CI/CD OS build.

## Kickstart install the OS

```
virt-install \  
  --name guest1-rhel7 \  
  --memory 2048 --vcpus 2 --disk size=8 \  
  --location http://example.com/path/to/os \  
  --os-variant rhel7 \  
  --initrd-inject /path/to/ks.cfg \  
  --extra-args="ks=file:/ks.cfg console=tty0 console=ttyS0,115200n8"
```

## Setup QEMU guest agent on virtual machines

```
systemctl enable qemu-guest-agent
```

# Building a Windows Image from ISO

**Download** container-native-virtualization/virtio-win - [Red Hat Container Catalog](#).

This contains the VirtIO drivers for Windows.

```
podman login registry.redhat.io
podman pull registry.redhat.io/container-native-virtualization/virtio-win
```

## Create a image disk at least 15GB in size

```
qemu-img create -f qcow2 w2016.qcow2 15G
```

## Install Windows using virt-install

```
virt-install --connect qemu:///system \
  --name ws2016 --ram 4096 --vcpus 2 \
  --network network=default,model=virtio \
  --disk path=ws2016.qcow2,format=qcow2,device=disk,bus=sata \
  --cdrom Windows_Server.ISO --disk path=virtio-win-0.1.189.iso,device=cdrom \
  --vnc --os-type windows --os-variant win2k16
```

## Post-install steps

- Microsoft update, likely a few reboots required.
- Install QEMU guest agent and VirtIO Drivers.
- Configure RDP access, install cloud init and sysprep the image.



# Processing and converting images

## Sparsify then compress the image

```
virt-sparsify --in-place rhel7.qcow2  
qemu-img convert -O qcow2 -c w2016.qcow2 windows2016.qcow2  
qemu-img convert -O qcow2 -c r7.qcow2 rhel7.qcow2
```

## Create a SHA256 for your images

This is optional, but good practice when uploading your images to a webserver, etc.

```
sha256sum *qcow2 > SHA256SUMS
```

## Optionally, sign your image with GPG.

```
gpg --sign myfile
```

# Creating and pushing images to the container registry

## Create a Dockerfile

```
FROM scratch
ADD windows2016.qcow2 /disk/
```

## Create a container

```
podman build -t cmihai/windows2016 .
```

## Login to the container registry

```
REGISTRY="$(oc get route/default-route \
  -n openshift-image-registry -o=jsonpath='{.spec.host}')"
podman login ${REGISTRY}
```

## Tag and push the image to your desired namespace (ex: virtual-machines)

```
podman tag localhost/virtual-machine/windows2016 \
  ${REGISTRY}/virtual-machines/windows2016
podman push ${REGISTRY}/virtual-machines/windows2016
```

# Creating container images with Buildah

## Create a Dockerfile in /tmp/vmdisk

```
cat << END > Dockerfile
FROM kubevirt/container-disk-v1alpha
ADD fedora34.qcow2 /disk
END
```

## Build and push to registry

```
buildah bud -t vmdisk/fedora34:latest /tmp/vmdisk
buildah push --tls-verify=false \
    vmdisk/fedora34:latest \
    docker://cdi-docker-registry-host.cdi/fedora34:latest
```

## Import the registry image into a Data volume

### YAML

```
apiVersion: cdi.kubevirt.io/v1alpha1
kind: DataVolume
metadata:
  name: fedora34image
spec:
  source:
    registry:
      url: "docker://image-registry.openshift-image-registry.svc:5000/
          virtual-machines/fedora34"
  pvc:
    accessModes:
      - ReadWriteMany
    resources:
      requests:
        storage: 20Gi
```

### Get the image info

```
oc apply -f datavolume.yaml
oc get pvc, dvs, pods # look for importer-fedora34image
```

# Uploading local disk images by using the virtctl tool

## Creating an upload DataVolume YAML

```
apiVersion: cdi.kubevirt.io/v1alpha1
kind: DataVolume
metadata:
  name: <upload-datavolume>
spec:
  source:
    upload: {}
  pvc:
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
        storage: <2Gi>
```

## Create the volume

```
oc create -f <upload-datavolume>.yaml
```

## Upload the image

```
virtctl image-upload dv <volume_name> \
  --size=2G \
  --image-path=</path/to/image>
```

## Verify that a DataVolume was created

## View all DataVolume objects

```
oc get dvs
```

@CiriVetiMihai

- To reduce costs, you can de-provision the build host after running a build.
- There may be trade-offs here: running costs with elastic provisioning vs. reserved instances.

```
ibmcloud sl hardware cancel IDENTIFIER --immediate
```

### **3) Automate the OS build steps with Ansible**

---

# Automation process overview

## Considerations

- Do we need to have this always available?
- Create, build and tear down the build host on demand - will also reduce costs.
- Are we build images for other hypervisors? For example, we can install VMware workstation and build VMware images as well.

## Build Process Overview

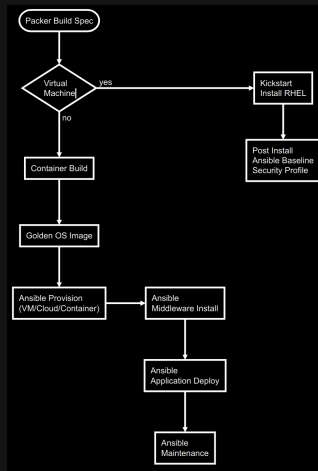


Figure 3: Automation Process Overview



# Setting up the build host on demand with Ansible

## Using an Ansible collection to install Virtualization Tools

- Created a Ansible collection for various virtualization tools (KVM, VMware, VirtualBox, etc).
- Collections make maintaining, publishing and testing roles easier.
- A simple example on how such a role can be used is listed below.
- This will install KVM, VMware, Packer and Vagrant on our build host.

## Install the Ansible collection

```
pip install --upgrade ansible
ansible-galaxy collection install \
    crivetimihai.virtualization
```

## Run the playbook

```
ansible-playbook -i localhost, playbook.yml \
    -e "vmware_workstation_license_key='...'" \
    -e "ansible_python_interpreter=/usr/bin/python3"
```

## playbook.yaml

```
- name: setup a virtualization environment
  hosts: all
  connection: local
  become: yes
  gather_facts: yes
  roles:
    - role: crivetimihai.virtualization.kvm
    - role: crivetimihai.virtualization.vmware
    - role: crivetimihai.virtualization.packer
    - role: crivetimihai.virtualization.vagrant
```

## What's inside the KVM role?

### vars/main.yaml

```
redhat_packages:
  - policycoreutils-python-utils
  - qemu-kvm
  - qemu-img
  - libvirt
  - libvirt-client
  - python3-libvirt
  - libvirt-nss
  - libguestfs-tools
  - virt-install
  - virt-top
  - genisoimage
```

### tasks/install\_RedHat.yml

```
- name: install RedHat packages
  package:
    name: "{{ redhat_packages }}"
    state: present
  become: yes
```

## Installing the correct VM tools

Automate the installation of the correct 'VM' tools (KVM, VMware, VirtualBox, etc) based on the detected hypervisor.

[https://github.com/crivetimiha/ansible\\_virtualization/blob/master/roles/vmtools/vars/main.yml](https://github.com/crivetimiha/ansible_virtualization/blob/master/roles/vmtools/vars/main.yml)

### tasks/main.yml

```
# VMware
- include: VMware.yml
  when: ansible_virtualization_type == "vmware"

# KVM
- include: KVM.yml
  when: ansible_virtualization_type == "kvm"
```

### tasks/KVM.yml

```
- name: install KVM tools packages
  package:
    name: "{{ packages_kvm_tools }}"
    state: present
  become: yes
```

**4) Automate the OS install and trigger Ansible post-install steps with Packer**

---

# Packer: build multiple images from a single source



Figure 4: Packer flow

## Variables to parametrized builds and hide secrets

```
{  
  "variables": {  
    "my_secret": "{{env `MY_SECRET`}}",  
    "not_a_secret": "plaintext",  
    "foo": "bar"  
  },  
  
  "sensitive-variables": ["my_secret", "foo"],  
}
```

## KVM Builder (qemu)

```
"builders": [{
  "type": "qemu",
  "accelerator": "kvm",
  "format": "qcow2",
  "disk_interface": "virtio-scsi",
  "boot_command": [
    "<up><wait><tab>",
    " text inst.ks=http://{{ .HTTPIP }}:{{ .HTTPPort }}/
    {user `vm_name`}.cfg net.ifnames=0",
    "<enter><wait>"
  ],
  "vm_name": "{{split build_type \"-\" 0}}-{{user `vm_name`}}",
  "iso_urls": [
    "{{user `iso_local_url`}}",
    "{{user `iso_download_url`}}"
  ],
  "iso_checksum": "{{user `iso_sha256`}}",
  "output_directory": "{{user `builds_dir`}}/{{split build_type \"-\" 0}}
  -{{user `vm_name`}}", ]}
```

## Provisioners: run post-install tasks

### Chaining multiple provisioners

```
"provisioners": [  
  {  
    "type": "shell",  
    "script": "setup.sh"  
  },  
  {  
    "type": "ansible",  
    "playbook_file": "{{user `playbook_file`}}"  
  }  
],
```



## Post-processors: compress or upload your image

### Compress, post-process and upload the results

```
{
  "post-processors": [
    {
      "type": "compress",
      "format": "tar.gz"
    },
    {
      "type": "upload",
      "endpoint": "http://example.com"
    }
  ]
}
```

# Building a VirtualBox image for RHEL 8 using Kickstart

virtualbox-iso output will be in this color.

```
==> virtualbox-iso: Retrieving Guest additions
==> virtualbox-iso: Trying ../../iso/VBoxGuestAdditions_6.0.10.iso
==> virtualbox-iso: Trying ../../iso/VBoxGuestAdditions_6.0.10.iso?checksum=sha256%3Ac8a686f8c7ad9ca8375961ab19815cec6b1f0d2496900a356a38ce86fe8a1325
==> virtualbox-iso: ../../iso/VBoxGuestAdditions_6.0.10.iso?checksum=sha256%3Ac8a686f8c7ad9ca8375961ab19815cec6b1f0d2496900a356a38ce86fe8a1325 => /Users/cmihai/github/packer-rhel-8/builds/../../iso/VBoxGuestAdditions_6.0.10.iso?checksum=sha256%3Ac8a686f8c7ad9ca8375961ab19815cec6b1f0d2496900a356a38ce86fe8a1325
==> virtualbox-iso: leaving retrieve loop for Guest additions
==> virtualbox-iso: Retrieving ISO
==> virtualbox-iso: Trying ../../iso/rhel-8.0-x86_64-dvd.iso
==> virtualbox-iso: Trying ../../iso/rhel-8.0-x86_64-dvd.iso?checksum=sha256%3Ade7be63802cc6cc43002cc6cc43
==> virtualbox-iso: ../../iso/rhel-8.0-x86_64-dvd.iso?checksum=sha256%3Ade7be63802cc6cc43002cc6cc43 => /Users/cmihai/github/packer-rhel-8/builds/../../iso/rhel-8.0-x86_64-dvd.iso?checksum=sha256%3Ade7be63802cc6cc43002cc6cc43
==> virtualbox-iso: leaving retrieve loop for ISO
==> virtualbox-iso: Starting HTTP server on port 8593
==> virtualbox-iso: Creating virtual machine...
==> virtualbox-iso: Creating hard drive...
==> virtualbox-iso: Creating forwarded port mapping for communicator (SSH, WinRM, etc) (host port 4344)
==> virtualbox-iso: Executing custom VBoxManage commands...
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --cpus 4
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --memory 4096
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --ioapic on
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --pae on
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --rtcuseutc on
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --bioslogodisplaytime 1
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --usb off
virtualbox-iso: Executing: modifyvm virtualbox-rhel-8-base --usbbehci off
==> virtualbox-iso: Starting the virtual machine...
virtualbox-iso: The VM will be run headless, without a GUI. If you want to
virtualbox-iso: view the screen of the VM, connect via VRDP without a password to
virtualbox-iso: rdp://127.0.0.1:5999
==> virtualbox-iso: Waiting 5s for boot...
==> virtualbox-iso: Typing the boot command...
==> virtualbox-iso: Using ssh communicator to connect: 127.0.0.1
==> virtualbox-iso: Waiting for SSH to become available...
```

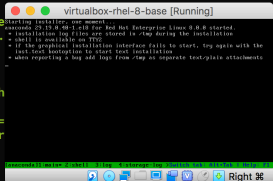


Figure 5: Packer

## **5) Automate Security and Compliance with OpenSCAP profiles**

---

# OpenSCAP security report

Title	Severity	Result
▼ Guide to the Secure Configuration of Red Hat Enterprise Linux 7 31x fail 4x notchecked		
▼ Services 7x fail		
▶ Obsolete Services		
▼ Base Services 1x fail		
Disable Automatic Bug Reporting Tool (abrt)	unknown	fail
▼ SSH Server 6x fail		
▼ Configure OpenSSH Server if Necessary 6x fail		
Disable SSH Access via Empty Passwords	high	pass
Set SSH Client Alive Count	medium	fail
Set SSH Idle Timeout Interval	unknown	fail
Enable SSH Warning Banner	medium	fail
Do Not Allow SSH Environment Options	medium	fail
Allow Only SSH Protocol 2	high	pass
Disable SSH Support for .rhosts Files	medium	pass
Use Only FIPS 140-2 Validated Ciphers	medium	fail
Disable Host-Based Authentication	medium	pass
Enable SSH Server firewalld Firewall exception	unknown	fail
Disable SSH Root Login	medium	pass

@CriVetiMihai

# Automatic Remediation as shell, ansible or puppet

Remediation Shell script: [\(show\)](#)

Remediation Ansible snippet: [\(show\)](#)

Complexity:	low
Disruption:	low
Strategy:	disable

```
- name: Disable service abrttd
  service:
    name: "{{item}}"
    enabled: "no"
    state: "stopped"
  register: service_result
  failed_when: "service_result is failed and ('Could not find the requested service' not in service_result.msg)"
  with_items:
    - abrttd
  tags:
    - service_abrttd_disabled
    - unknown_severity
    - disable_strategy
    - low_complexity
    - low_disruption
    - NIST-800-53-AC-17(8)
    - NIST-800-53-CM-7
```

Figure 7: OpenSCAP Remediation

## Install OpenSCAP

```
dnf install openscap-scanner
```

## Generate a report

```
sudo oscap xccdf eval --report report.html \  
  --profile xccdf_org.ssgproject.content_profile_pci-dss \  
  /usr/share/xml/scap/ssg/content/ssg-rhel7-ds.xml
```

## **6) Test your Ansible Playbooks with Molecule**

---

## Creating a vagrant or docker machine and trigger goss tests:

```
molecule create -s vagrant-rhel-8  
molecule converge -s vagrant-rhel-8  
molecule login
```

## In one step

```
molecule test
```

## Another OS:

```
molecule create -s docker-rhel-7
```



## molecule.yml with Fedora 34 running on Docker

```
driver:  
  name: docker  
  provider:  
    name: docker  
lint:  
  name: yamllint  
platforms:  
  - name: fedora-34  
    image: fedora:34  
    dockerfile: ../resources/Dockerfile.j2  
provisioner:  
  name: ansible
```

## 7) Templates with cookiecutter

---

## Cookiecutter: Better Project Templates

- Cookiecutter creates projects from project templates, e.g. Ansible role structure, with molecule tests.
- Molecule provides a native cookiecutter interface, so developers can provide their own templates.

## Create a new role from a template, with molecule tests included

```
molecule init template \  
  --url https://github.com/crivetimihai/ansible_cookiecutter.git \  
  --role-name httpd
```

## **8) Orchestrate Workflows with Ansible Tower**

---

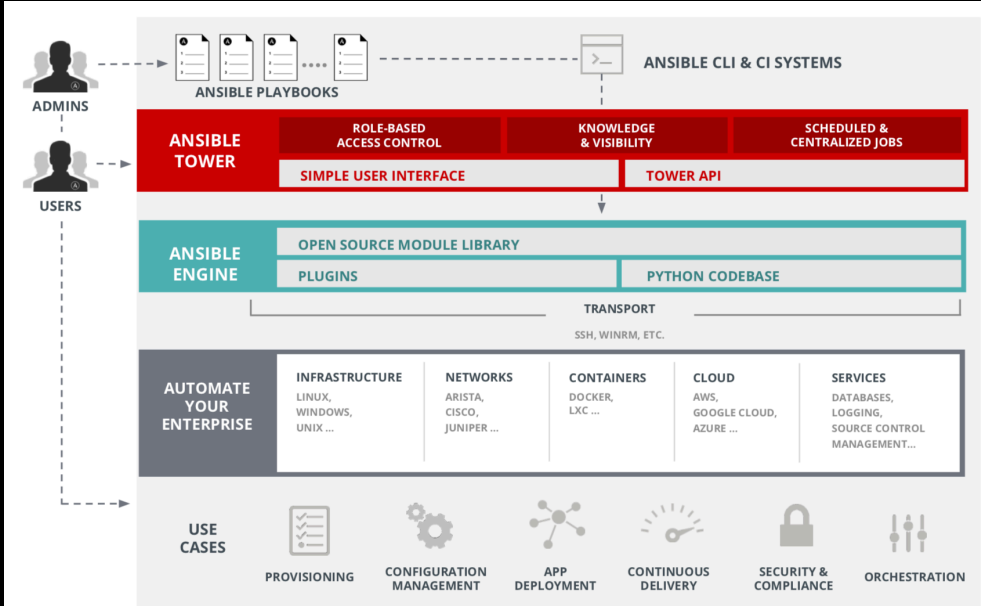
Ansible Tower is a UI and RESTful API allowing you to scale IT automation, manage complex deployments and speed productivity.

- Role-based access control
- Deploy entire applications with push-button deployment access
- All automations are centrally logged
- Powerful workflows match your IT processes



**Figure 8:** Ansible Tower

# Ansible for Enterprise: Architecture



## **99) Resources, Questions and Answers**

---

### Tools and roles

- <https://github.com/crivetimihaipacker-os-image-builder>
- <https://devopstoolbox.github.io/docs/>
- <https://galaxy.ansible.com/crivetimihaivirtualization> Virtualization Collection

### Other useful roles

- Kubernetes Collection for Ansible [https://docs.openshift.com/container-platform/4.8/operators/operator\\_sdk/ansible/osdk-ansible-k8s-collection.html](https://docs.openshift.com/container-platform/4.8/operators/operator_sdk/ansible/osdk-ansible-k8s-collection.html)
- <https://github.com/RedHatGov/ansible-kvm-vm>
- <https://github.com/CSCfi/ansible-role-provision-vm>



**Thank you, and please reach out and follow me on:**

- @CrivetiMihai
- <https://www.linkedin.com/in/crivetimihai/>
- <http://blog.boreas.ro/>