Red Hat OpenStack Platform 16.1

Transitioning to Containerized Services

A basic guide to working with OpenStack Platform containerized services

Last Updated: 2020-09-25

Red Hat OpenStack Platform 16.1 Transitioning to Containerized Services

A basic guide to working with OpenStack Platform containerized services

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

This guide provides some basic information to help users get accustomed working with OpenStack Platform services running in containers.

Table of Contents

CHAPTER 1. INTRODUCTION

Past versions of Red Hat OpenStack Platform used services managed with Systemd. However, more recent version of OpenStack Platform now use containers to run services. Some administrators might not have a good understanding of how containerized OpenStack Platform services operate, and so this guide aims to help you understand OpenStack Platform container images and containerized services. This includes:

- How to obtain and modify container images
- How to manage containerized services in the overcloud
- Understanding how containers differ from Systemd services

The main goal is to help you gain enough knowledge of containerized OpenStack Platform services to transition from a Systemd-based environment to a container-based environment.

1.1. CONTAINERIZED SERVICES AND KOLLA

Each of the main Red Hat OpenStack Platform services run in containers. This provides a method of keep each service within its own isolated namespace separated from the host. This means:

- The deployment of services is performed by pulling container images from the Red Hat Custom Portal and running them.
- The management functions, like starting and stopping services, operate through the **podman** command.
- Upgrading containers require pulling new container images and replacing the existing containers with newer versions.

Red Hat OpenStack Platform uses a set of containers built and managed with the **kolla** toolset.

CHAPTER 2. OBTAINING AND MODIFYING CONTAINER IMAGES

A containerized overcloud requires access to a registry with the required container images. This chapter provides information on how to prepare the registry and your undercloud and overcloud configuration to use container images for Red Hat OpenStack Platform.

2.1. PREPARING CONTAINER IMAGES

The overcloud installation requires an environment file to determine where to obtain container images and how to store them. Generate and customize this environment file that you can use to prepare your container images.

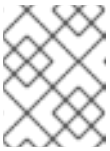
Procedure

1. Log in to your undercloud host as the **stack** user.
2. Generate the default container image preparation file:

```
$ openstack tripleo container image prepare default \
  --local-push-destination \
  --output-env-file containers-prepare-parameter.yaml
```

This command includes the following additional options:

- **--local-push-destination** sets the registry on the undercloud as the location for container images. With this option, director pulls the necessary images from the Red Hat Container Catalog and pushes the images to the registry on the undercloud. Director uses the undercloud registry as the container image source. To pull container images directly from the Red Hat Container Catalog, omit this option.
- **--output-env-file** specifies an environment file that includes include the parameters for preparing your container images. In this example, the name of the file is **containers-prepare-parameter.yaml**.



NOTE

You can use the same **containers-prepare-parameter.yaml** file to define a container image source for both the undercloud and the overcloud.

3. Modify the **containers-prepare-parameter.yaml** to suit your requirements.

2.2. CONTAINER IMAGE PREPARATION PARAMETERS

The default file for preparing your containers (**containers-prepare-parameter.yaml**) contains the **ContainerImagePrepare** heat parameter. This parameter defines a list of strategies for preparing a set of images:

```
parameter_defaults:
  ContainerImagePrepare:
  - (strategy one)
```

```
- (strategy two)
- (strategy three)
...
```

Each strategy accepts a set of sub-parameters that defines which images to use and what to do with the images. The following table contains information about the sub-parameters you can use with each **ContainerImagePrepare** strategy:

Parameter	Description
excludes	List of image name substrings to exclude from a strategy.
includes	List of image name substrings to include in a strategy. At least one image name must match an existing image. All **excludes** are ignored if **includes** is specified.
modify_append_tag	String to append to the tag for the destination image. For example, if you pull an image with the tag **14.0-89** and set the **modify_append_tag** to **-hotfix**, the director tags the final image as **14.0-89-hotfix**.
modify_only_with_labels	A dictionary of image labels that filter the images that you want to modify. If an image matches the labels defined, the director includes the image in the modification process.
modify_role	String of ansible role names to run during upload but before pushing the image to the destination registry.
modify_vars	Dictionary of variables to pass to **modify_role**.

Parameter	Description
push_destination	Defines the namespace of the registry that you want to push images to during the upload process. • If set to **true**, the **push_destination** is set to the undercloud registry namespace using the hostname, which is the recommended method. • If set to **false**, the push to a local registry does not occur and nodes pull images directly from the source. • If set to a custom value, director pushes images to an external local registry. Do not set this parameter to **false** in production environments. If the **push_destination** parameter is set to **false** or is not defined and the remote registry requires authentication, set the **ContainerImageRegistryLogin** parameter to **true** and include the credentials with the **ContainerImageRegistryCredentials** parameter.
pull_source	The source registry from where to pull the original container images.
set	A dictionary of **key: value** definitions that define where to obtain the initial images.
tag_from_label	Defines the label pattern to tag the resulting images. Usually sets to **{version}-{release}**.

IMPORTANT

When you push images to the undercloud, use **push_destination: true** instead of **push_destination: UNDERCLOUD_IP:PORT**. The **push_destination: true** method provides a level of consistency across both IPv4 and IPv6 addresses.

The **set** parameter accepts a set of **key: value** definitions:

Key	Description
ceph_image	The name of the Ceph Storage container image.
ceph_namespace	The namespace of the Ceph Storage container image.

Key	Description
ceph_tag	The tag of the Ceph Storage container image.
name_prefix	A prefix for each OpenStack service image.
name_suffix	A suffix for each OpenStack service image.
namespace	The namespace for each OpenStack service image.
neutron_driver	The driver to use to determine which OpenStack Networking (neutron) container to use. Use a null value to set to the standard **neutron-server** container. Set to **ovn** to use OVN-based containers.
tag	The tag that the director uses to identify the images to pull from the source registry. You usually keep this key set to the default value, which is the Red Hat OpenStack Platform version number.

NOTE

The container images use multi-stream tags based on Red Hat OpenStack Platform version. This means there is no longer a **latest** tag.

The **ContainerImageRegistryCredentials** parameter maps a container registry to a username and password to authenticate to that registry.

If a container registry requires a username and password, you can use **ContainerImageRegistryCredentials** to include credentials with the following syntax:

```
ContainerImagePrepare:
- push_destination: true
  set:
    namespace: registry.redhat.io/...
    ...
ContainerImageRegistryCredentials:
  registry.redhat.io:
    my_username: my_password
```

In the example, replace **my_username** and **my_password** with your authentication credentials. Instead of using your individual user credentials, Red Hat recommends creating a registry service account and using those credentials to access **registry.redhat.io** content. For more information, see "Red Hat Container Registry Authentication".

The **ContainerImageRegistryLogin** parameter is used to control the registry login on the systems being deployed. This must be set to **true** if **push_destination** is set to false or not used.

```
ContainerImagePrepare:
- set:
```

```
    namespace: registry.redhat.io/...
    ...
ContainerImageRegistryCredentials:
  registry.redhat.io:
    my_username: my_password
ContainerImageRegistryLogin: true
```

If you have configured **push_destination**, do not set **ContainerImageRegistryLogin** to **true**. If you set this option to **true** and the overcloud nodes do not have network connectivity to the registry hosts defined in **ContainerImageRegistryCredentials**, the deployment might fail when trying to perform a login.

2.3. LAYERING IMAGE PREPARATION ENTRIES

The value of the **ContainerImagePrepare** parameter is a YAML list. This means that you can specify multiple entries. The following example demonstrates two entries where director uses the latest version of all images except for the **nova-api** image, which uses the version tagged with **16.0-44**:

```
ContainerImagePrepare:
- tag_from_label: "{version}-{release}"
  push_destination: true
  excludes:
  - nova-api
  set:
    namespace: registry.redhat.io/rhosp-rhel8
    name_prefix: openstack-
    name_suffix: ''
    tag: 16.1
- push_destination: true
  includes:
  - nova-api
  set:
    namespace: registry.redhat.io/rhosp-rhel8
    tag: 16.1-44
```

The **includes** and **excludes** entries control image filtering for each entry. The images that match the **includes** strategy take precedence over **excludes** matches. The image name must contain the **includes** or **excludes** value to be considered a match.

2.4. MODIFYING IMAGES DURING PREPARATION

It is possible to modify images during image preparation, and then immediately deploy with modified images. Scenarios for modifying images include:

- As part of a continuous integration pipeline where images are modified with the changes being tested before deployment.
- As part of a development workflow where local changes must be deployed for testing and development.
- When changes must be deployed but are not available through an image build pipeline. For example, adding proprietary add-ons or emergency fixes.

To modify an image during preparation, invoke an Ansible role on each image that you want to modify. The role takes a source image, makes the requested changes, and tags the result. The prepare

command can push the image to the destination registry and set the heat parameters to refer to the modified image.

The Ansible role **tripleo-modify-image** conforms with the required role interface and provides the behaviour necessary for the modify use cases. Control the modification with the modify-specific keys in the **ContainerImagePrepare** parameter:

- **modify_role** specifies the Ansible role to invoke for each image to modify.
- **modify_append_tag** appends a string to the end of the source image tag. This makes it obvious that the resulting image has been modified. Use this parameter to skip modification if the **push_destination** registry already contains the modified image. Change **modify_append_tag** whenever you modify the image.
- **modify_vars** is a dictionary of Ansible variables to pass to the role.

To select a use case that the **tripleo-modify-image** role handles, set the **tasks_from** variable to the required file in that role.

While developing and testing the **ContainerImagePrepare** entries that modify images, run the image prepare command without any additional options to confirm that the image is modified as you expect:

```
sudo openstack tripleo container image prepare \
  -e ~/containers-prepare-parameter.yaml
```

2.5. UPDATING EXISTING PACKAGES ON CONTAINER IMAGES

The following example **ContainerImagePrepare** entry updates in all packages on the container images using the dnf repository configuration of the undercloud host:

```
ContainerImagePrepare:
- push_destination: true
  ...
  modify_role: tripleo-modify-image
  modify_append_tag: "-updated"
  modify_vars:
    tasks_from: yum_update.yml
    compare_host_packages: true
    yum_repos_dir_path: /etc/yum.repos.d
  ...
```

2.6. INSTALLING ADDITIONAL RPM FILES TO CONTAINER IMAGES

You can install a directory of RPM files in your container images. This is useful for installing hotfixes, local package builds, or any package that is not available through a package repository. For example, the following **ContainerImagePrepare** entry installs some hotfix packages only on the **nova-compute** image:

```
ContainerImagePrepare:
- push_destination: true
  ...
  includes:
  - nova-compute
  modify_role: tripleo-modify-image
```

```
  modify_append_tag: "-hotfix"
  modify_vars:
    tasks_from: rpm_install.yml
    rpms_path: /home/stack/nova-hotfix-pkgs
  ...
```

2.7. MODIFYING CONTAINER IMAGES WITH A CUSTOM DOCKERFILE

For maximum flexibility, you can specify a directory containing a Dockerfile to make the required changes. When you invoke the **tripleo-modify-image** role, the role generates a **Dockerfile.modified** file that changes the **FROM** directive and adds extra **LABEL** directives. The following example runs the custom Dockerfile on the **nova-compute** image:

```
ContainerImagePrepare:
- push_destination: true
  ...
  includes:
  - nova-compute
  modify_role: tripleo-modify-image
  modify_append_tag: "-hotfix"
  modify_vars:
    tasks_from: modify_image.yml
    modify_dir_path: /home/stack/nova-custom
  ...
```

The following example shows the **/home/stack/nova-custom/Dockerfile** file. After you run any **USER** root directives, you must switch back to the original image default user:

```
FROM registry.redhat.io/rhosp-rhel8/openstack-nova-compute:latest

USER "root"

COPY customize.sh /tmp/
RUN /tmp/customize.sh

USER "nova"
```

2.8. PREPARING A SATELLITE SERVER FOR CONTAINER IMAGES

Red Hat Satellite 6 offers registry synchronization capabilities. This provides a method to pull multiple images into a Satellite server and manage them as part of an application life cycle. The Satellite also acts as a registry for other container-enabled systems to use. For more information about managing container images, see Managing Container Images in the *Red Hat Satellite 6 Content Management Guide*.

The examples in this procedure use the **hammer** command line tool for Red Hat Satellite 6 and an example organization called **ACME**. Substitute this organization for your own Satellite 6 organization.

NOTE

This procedure requires authentication credentials to access container images from **registry.redhat.io**. Instead of using your individual user credentials, Red Hat recommends creating a registry service account and using those credentials to access **registry.redhat.io** content. For more information, see "Red Hat Container Registry Authentication".

Procedure

1. Create a list of all container images:

```
$ sudo podman search --limit 1000 "registry.redhat.io/rhosp" | grep rhosp-rhel8 | awk '{ print $2 }' | grep -v beta | sed "s/registry.redhat.io\///g" | tail -n+2 > satellite_images
```

2. Copy the **satellite_images** file to a system that contains the Satellite 6 **hammer** tool. Alternatively, use the instructions in the *Hammer CLI Guide* to install the **hammer** tool to the undercloud.

3. Run the following **hammer** command to create a new product (**OSP16.1 Containers**) in your Satellite organization:

```
$ hammer product create \
  --organization "ACME" \
  --name "OSP16.1 Containers"
```

   This custom product will contain your images.

4. Add the base container image to the product:

```
$ hammer repository create \
  --organization "ACME" \
  --product "OSP16.1 Containers" \
  --content-type docker \
  --url https://registry.redhat.io \
  --docker-upstream-name rhosp-rhel8/openstack-base \
  --upstream-username USERNAME \
  --upstream-password PASSWORD \
  --name base
```

5. Add the overcloud container images from the **satellite_images** file:

```
$ while read IMAGE; do \
  IMAGENAME=$(echo $IMAGE | cut -d"/" -f2 | sed "s/openstack-//g" | sed "s/:.*//g") ; \
  hammer repository create \
  --organization "ACME" \
  --product "OSP16.1 Containers" \
  --content-type docker \
  --url https://registry.redhat.io \
  --docker-upstream-name $IMAGE \
  --upstream-username USERNAME \
  --upstream-password PASSWORD \
  --name $IMAGENAME ; done < satellite_images
```

6. Add the Ceph Storage 4 container image:

```
$ hammer repository create \
  --organization "ACME" \
  --product "OSP16.1 Containers" \
  --content-type docker \
  --url https://registry.redhat.io \
  --docker-upstream-name rhceph-beta/rhceph-4-rhel8 \
  --upstream-username USERNAME \
  --upstream-password PASSWORD \
  --name rhceph-4-rhel8
```

7. Synchronize the container images:

```
$ hammer product synchronize \
  --organization "ACME" \
  --name "OSP16.1 Containers"
```

Wait for the Satellite server to complete synchronization.

NOTE

Depending on your configuration, **hammer** might ask for your Satellite server username and password. You can configure **hammer** to automatically login using a configuration file. For more information, see the Authentication section in the *Hammer CLI Guide*.

8. If your Satellite 6 server uses content views, create a new content view version to incorporate the images and promote it along environments in your application life cycle. This largely depends on how you structure your application lifecycle. For example, if you have an environment called **production** in your lifecycle and you want the container images to be available in that environment, create a content view that includes the container images and promote that content view to the **production** environment. For more information, see Managing Content Views.

9. Check the available tags for the **base** image:

```
$ hammer docker tag list --repository "base" \
  --organization "ACME" \
  --environment "production" \
  --content-view "myosp16_1" \
  --product "OSP16.1 Containers"
```

This command displays tags for the OpenStack Platform container images within a content view for a particular environment.

10. Return to the undercloud and generate a default environment file that prepares images using your Satellite server as a source. Run the following example command to generate the environment file:

```
(undercloud) $ openstack tripleo container image prepare default \
  --output-env-file containers-prepare-parameter.yaml
```

- **--output-env-file** is an environment file name. The contents of this file include the parameters for preparing your container images for the undercloud. In this case, the name of the file is **containers-prepare-parameter.yaml**.

11. Edit the **containers-prepare-parameter.yaml** file and modify the following parameters:

    - **namespace** - The URL and port of the registry on the Satellite server. The default registry port on Red Hat Satellite is 5000.

    - **name_prefix** - The prefix is based on a Satellite 6 convention. This differs depending on whether you use content views:

        - If you use content views, the structure is **[org]-[environment]-[content view]-[product]-**. For example: **acme-production-myosp16-osp16_containers-**.

        - If you do not use content views, the structure is **[org]-[product]-**. For example: **acme-osp16_1_containers-**.

    - **ceph_namespace**, **ceph_image**, **ceph_tag** - If you use Ceph Storage, include these additional parameters to define the Ceph Storage container image location. Note that **ceph_image** now includes a Satellite-specific prefix. This prefix is the same value as the **name_prefix** option.

The following example environment file contains Satellite-specific parameters:

```
parameter_defaults:
  ContainerImagePrepare:
  - push_destination: true
    set:
      ceph_image: acme-production-myosp16_1-osp16_1_containers-rhceph-4
      ceph_namespace: satellite.example.com:5000
      ceph_tag: latest
      name_prefix: acme-production-myosp16_1-osp16_1_containers-
      name_suffix: ''
      namespace: satellite.example.com:5000
      neutron_driver: null
      tag: 16.1
      ...
    tag_from_label: '{version}-{release}'
```

Use this environment file when you create both your undercloud and overcloud.

CHAPTER 3. INSTALLING THE UNDERCLOUD WITH CONTAINERS

This chapter provides info on how to create a container-based undercloud and keep it updated.

3.1. CONFIGURING DIRECTOR

The director installation process requires certain settings in the **undercloud.conf** configuration file, which director reads from the home directory of the **stack** user. Complete the following steps to copy default template as a foundation for your configuration.

Procedure

1. Copy the default template to the home directory of the **stack** user's:

```
[stack@director ~]$ cp \
  /usr/share/python-tripleoclient/undercloud.conf.sample \
  ~/undercloud.conf
```

2. Edit the **undercloud.conf** file. This file contains settings to configure your undercloud. If you omit or comment out a parameter, the undercloud installation uses the default value.

3.2. DIRECTOR CONFIGURATION PARAMETERS

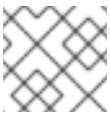
The following list contains information about parameters for configuring the **undercloud.conf** file. Keep all parameters within their relevant sections to avoid errors.

Defaults

The following parameters are defined in the **[DEFAULT]** section of the **undercloud.conf** file:

additional_architectures

A list of additional (kernel) architectures that an overcloud supports. Currently the overcloud supports **ppc64le** architecture.



NOTE

When you enable support for ppc64le, you must also set **ipxe_enabled** to **False**

certificate_generation_ca

The **certmonger** nickname of the CA that signs the requested certificate. Use this option only if you have set the **generate_service_certificate** parameter. If you select the **local** CA, certmonger extracts the local CA certificate to **/etc/pki/ca-trust/source/anchors/cm-local-ca.pem** and adds the certificate to the trust chain.

clean_nodes

Defines whether to wipe the hard drive between deployments and after introspection.

cleanup

Cleanup temporary files. Set this to **False** to leave the temporary files used during deployment in place after you run the deployment command. This is useful for debugging the generated files or if errors occur.

container_cli

The CLI tool for container management. Leave this parameter set to **podman**. Red Hat Enterprise Linux 8.2 only supports **podman**.

container_healthcheck_disabled

Disables containerized service health checks. Red Hat recommends that you enable health checks and leave this option set to **false**.

container_images_file

Heat environment file with container image information. This file can contain the following entries:

- Parameters for all required container images
- The **ContainerImagePrepare** parameter to drive the required image preparation. Usually the file that contains this parameter is named **containers-prepare-parameter.yaml**.

container_insecure_registries

A list of insecure registries for **podman** to use. Use this parameter if you want to pull images from another source, such as a private container registry. In most cases, **podman** has the certificates to pull container images from either the Red Hat Container Catalog or from your Satellite server if the undercloud is registered to Satellite.

container_registry_mirror

An optional **registry-mirror** configured that **podman** uses.

custom_env_files

Additional environment files that you want to add to the undercloud installation.

deployment_user

The user who installs the undercloud. Leave this parameter unset to use the current default user **stack**.

discovery_default_driver

Sets the default driver for automatically enrolled nodes. Requires the **enable_node_discovery** parameter to be enabled and you must include the driver in the **enabled_hardware_types** list.

enable_ironic; enable_ironic_inspector; enable_mistral; enable_nova; enable_tempest; enable_validations; enable_zaqar

Defines the core services that you want to enable for director. Leave these parameters set to **true**.

enable_node_discovery

Automatically enroll any unknown node that PXE-boots the introspection ramdisk. New nodes use the **fake_pxe** driver as a default but you can set **discovery_default_driver** to override. You can also use introspection rules to specify driver information for newly enrolled nodes.

enable_novajoin

Defines whether to install the **novajoin** metadata service in the undercloud.

enable_routed_networks

Defines whether to enable support for routed control plane networks.

enable_swift_encryption

Defines whether to enable Swift encryption at-rest.

enable_telemetry

Defines whether to install OpenStack Telemetry services (gnocchi, aodh, panko) in the undercloud. Set the **enable_telemetry** parameter to **true** if you want to install and configure telemetry services automatically. The default value is **false**, which disables telemetry on the undercloud. This parameter is required if you use other products that consume metrics data, such as Red Hat CloudForms.

enabled_hardware_types

A list of hardware types that you want to enable for the undercloud.

generate_service_certificate

Defines whether to generate an SSL/TLS certificate during the undercloud installation, which is used for the **undercloud_service_certificate** parameter. The undercloud installation saves the resulting certificate **/etc/pki/tls/certs/undercloud-[undercloud_public_vip].pem**. The CA defined in the **certificate_generation_ca** parameter signs this certificate.

heat_container_image

URL for the heat container image to use. Leave unset.

heat_native

Run host-based undercloud configuration using **heat-all**. Leave as **true**.

hieradata_override

Path to **hieradata** override file that configures Puppet hieradata on the director, providing custom configuration to services beyond the **undercloud.conf** parameters. If set, the undercloud installation copies this file to the **/etc/puppet/hieradata** directory and sets it as the first file in the hierarchy. For more information about using this feature, see Configuring hieradata on the undercloud.

inspection_extras

Defines whether to enable extra hardware collection during the inspection process. This parameter requires the **python-hardware** or **python-hardware-detect** packages on the introspection image.

inspection_interface

The bridge that director uses for node introspection. This is a custom bridge that the director configuration creates. The **LOCAL_INTERFACE** attaches to this bridge. Leave this as the default **br-ctlplane**.

inspection_runbench

Runs a set of benchmarks during node introspection. Set this parameter to **true** to enable the benchmarks. This option is necessary if you intend to perform benchmark analysis when inspecting the hardware of registered nodes.

ipa_otp

Defines the one-time password to register the undercloud node to an IPA server. This is required when **enable_novajoin** is enabled.

ipv6_address_mode

IPv6 address configuration mode for the undercloud provisioning network. The following list contains the possible values for this parameter:

- dhcpv6-stateless - Address configuration using router advertisement (RA) and optional information using DHCPv6.
- dhcpv6-stateful - Address configuration and optional information using DHCPv6.

ipxe_enabled

Defines whether to use iPXE or standard PXE. The default is **true**, which enables iPXE. Set this parameter to **false** to use standard PXE.

local_interface

The chosen interface for the director Provisioning NIC. This is also the device that director uses for DHCP and PXE boot services. Change this value to your chosen device. To see which device is connected, use the **ip addr** command. For example, this is the result of an **ip addr** command:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
```

```
1000
    link/ether 52:54:00:75:24:09 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.178/24 brd 192.168.122.255 scope global dynamic eth0
       valid_lft 3462sec preferred_lft 3462sec
    inet6 fe80::5054:ff:fe75:2409/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noop state DOWN
    link/ether 42:0b:c2:a5:c1:26 brd ff:ff:ff:ff:ff:ff
```

In this example, the External NIC uses **eth0** and the Provisioning NIC uses **eth1**, which is currently not configured. In this case, set the **local_interface** to **eth1**. The configuration script attaches this interface to a custom bridge defined with the **inspection_interface** parameter.

local_ip

The IP address defined for the director Provisioning NIC. This is also the IP address that director uses for DHCP and PXE boot services. Leave this value as the default **192.168.24.1/24** unless you use a different subnet for the Provisioning network, for example, if this IP address conflicts with an existing IP address or subnet in your environment.

local_mtu

The maximum transmission unit (MTU) that you want to use for the **local_interface**. Do not exceed 1500 for the undercloud.

local_subnet

The local subnet that you want to use for PXE boot and DHCP interfaces. The **local_ip** address should reside in this subnet. The default is **ctlplane-subnet**.

net_config_override

Path to network configuration override template. If you set this parameter, the undercloud uses a JSON format template to configure the networking with **os-net-config** and ignores the network parameters set in **undercloud.conf**. Use this parameter when you want to configure bonding or add an option to the interface. See **/usr/share/instack-undercloud/templates/net-config.json.template** for an example.

networks_file

Networks file to override for **heat**.

output_dir

Directory to output state, processed heat templates, and Ansible deployment files.

overcloud_domain_name

The DNS domain name that you want to use when you deploy the overcloud.

NOTE

When you configure the overcloud, you must set the **CloudDomain** parameter to a matching value. Set this parameter in an environment file when you configure your overcloud.

roles_file

The roles file that you want to use to override the default roles file for undercloud installation. It is highly recommended to leave this parameter unset so that the director installation uses the default roles file.

scheduler_max_attempts

The maximum number of times that the scheduler attempts to deploy an instance. This value must be greater or equal to the number of bare metal nodes that you expect to deploy at once to avoid potential race conditions when scheduling.

service_principal

The Kerberos principal for the service using the certificate. Use this parameter only if your CA requires a Kerberos principal, such as in FreeIPA.

subnets

List of routed network subnets for provisioning and introspection. The default value includes only the **ctlplane-subnet** subnet. For more information, see Subnets.

templates

Heat templates file to override.

undercloud_admin_host

The IP address or hostname defined for director Admin API endpoints over SSL/TLS. The director configuration attaches the IP address to the director software bridge as a routed IP address, which uses the **/32** netmask.

undercloud_debug

Sets the log level of undercloud services to **DEBUG**. Set this value to **true** to enable **DEBUG** log level.

undercloud_enable_selinux

Enable or disable SELinux during the deployment. It is highly recommended to leave this value set to **true** unless you are debugging an issue.

undercloud_hostname

Defines the fully qualified host name for the undercloud. If set, the undercloud installation configures all system host name settings. If left unset, the undercloud uses the current host name, but you must configure all system host name settings appropriately.

undercloud_log_file

The path to a log file to store the undercloud install and upgrade logs. By default, the log file is **install-undercloud.log** in the home directory. For example, **/home/stack/install-undercloud.log**.

undercloud_nameservers

A list of DNS nameservers to use for the undercloud hostname resolution.

undercloud_ntp_servers

A list of network time protocol servers to help synchronize the undercloud date and time.

undercloud_public_host

The IP address or hostname defined for director Public API endpoints over SSL/TLS. The director configuration attaches the IP address to the director software bridge as a routed IP address, which uses the **/32** netmask.

undercloud_service_certificate

The location and filename of the certificate for OpenStack SSL/TLS communication. Ideally, you obtain this certificate from a trusted certificate authority. Otherwise, generate your own self-signed certificate.

undercloud_timezone

Host timezone for the undercloud. If you do not specify a timezone, director uses the existing timezone configuration.

undercloud_update_packages

Defines whether to update packages during the undercloud installation.

Subnets

Each provisioning subnet is a named section in the **undercloud.conf** file. For example, to create a subnet called **ctlplane-subnet**, use the following sample in your **undercloud.conf** file:

```
[ctlplane-subnet]
cidr = 192.168.24.0/24
dhcp_start = 192.168.24.5
dhcp_end = 192.168.24.24
inspection_iprange = 192.168.24.100,192.168.24.120
gateway = 192.168.24.1
masquerade = true
```

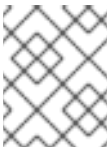
You can specify as many provisioning networks as necessary to suit your environment.

cidr

The network that director uses to manage overcloud instances. This is the Provisioning network, which the undercloud **neutron** service manages. Leave this as the default **192.168.24.0/24** unless you use a different subnet for the Provisioning network.

masquerade

Defines whether to masquerade the network defined in the **cidr** for external access. This provides the Provisioning network with a degree of network address translation (NAT) so that the Provisioning network has external access through director.



NOTE

The director configuration also enables IP forwarding automatically using the relevant **sysctl** kernel parameter.

dhcp_start; dhcp_end

The start and end of the DHCP allocation range for overcloud nodes. Ensure that this range contains enough IP addresses to allocate your nodes.

dhcp_exclude

IP addresses to exclude in the DHCP allocation range.

dns_nameservers

DNS nameservers specific to the subnet. If no nameservers are defined for the subnet, the subnet uses nameservers defined in the **undercloud_nameservers** parameter.

gateway

The gateway for the overcloud instances. This is the undercloud host, which forwards traffic to the External network. Leave this as the default **192.168.24.1** unless you use a different IP address for director or want to use an external gateway directly.

host_routes

Host routes for the Neutron-managed subnet for the overcloud instances on this network. This also configures the host routes for the **local_subnet** on the undercloud.

inspection_iprange

Temporary IP range for nodes on this network to use during the inspection process. This range must not overlap with the range defined by **dhcp_start** and **dhcp_end** but must be in the same IP subnet.

Modify the values for these parameters to suit your configuration. When complete, save the file.

3.3. INSTALLING DIRECTOR

Complete the following steps to install director and perform some basic post-installation tasks.

Procedure

1. Run the following command to install director on the undercloud:

```
[stack@director ~]$ openstack undercloud install
```

   This command launches the director configuration script. Director installs additional packages and configures its services according to the configuration in the **undercloud.conf**. This script takes several minutes to complete.

   The script generates two files:

   - **undercloud-passwords.conf** - A list of all passwords for the director services.
   - **stackrc** - A set of initialization variables to help you access the director command line tools.

2. The script also starts all OpenStack Platform service containers automatically. You can check the enabled containers with the following command:

```
[stack@director ~]$ sudo podman ps
```

3. To initialize the **stack** user to use the command line tools, run the following command:

```
[stack@director ~]$ source ~/stackrc
```

   The prompt now indicates that OpenStack commands authenticate and execute against the undercloud;

```
(undercloud) [stack@director ~]$
```

The director installation is complete. You can now use the director command line tools.

3.4. PERFORMING A MINOR UPDATE OF A CONTAINERIZED UNDERCLOUD

The director provides commands to update the packages on the undercloud node. This allows you to perform a minor update within the current version of your OpenStack Platform environment.

Procedure

1. Log in to the director as the **stack** user.
2. Run **dnf** to upgrade the director's main packages:

```
$ sudo dnf update -y python3-tripleoclient* openstack-tripleo-common openstack-tripleo-heat-templates tripleo-ansible ansible
```

3. The director uses the **openstack undercloud upgrade** command to update the undercloud environment. Run the command:

```
$ openstack undercloud upgrade
```

4. Wait until the undercloud upgrade process completes.
5. Reboot the undercloud to update the operating system's kernel and other system packages:

```
$ sudo reboot
```

6. Wait until the node boots.

CHAPTER 4. DEPLOYING AND UPDATING AN OVERCLOUD WITH CONTAINERS

This chapter provides info on how to create a container-based overcloud and keep it updated.

4.1. DEPLOYING AN OVERCLOUD

This procedure demonstrates how to deploy an overcloud with minimum configuration. The result will be a basic two-node overcloud (1 Controller node, 1 Compute node).

Procedure

1. Source the **stackrc** file:

```
$ source ~/stackrc
```

2. Run the **deploy** command and include the file containing your overcloud image locations (usually **overcloud_images.yaml**):

```
(undercloud) $ openstack overcloud deploy --templates \
  -e /home/stack/templates/overcloud_images.yaml \
  --ntp-server pool.ntp.org
```

3. Wait until the overcloud completes deployment.

4.2. UPDATING AN OVERCLOUD

For information on updating a containerized overcloud, see the Keeping Red Hat OpenStack Platform Updated guide.

CHAPTER 5. WORKING WITH CONTAINERIZED SERVICES

This chapter provides some examples of commands to manage containers and how to troubleshoot your OpenStack Platform containers

5.1. MANAGING CONTAINERIZED SERVICES

Red Hat OpenStack Platform (RHOSP) runs services in containers on the undercloud and overcloud nodes. In certain situations, you might need to control the individual services on a host. This section contains information about some common commands you can run on a node to manage containerized services.

Listing containers and images

To list running containers, run the following command:

```
$ sudo podman ps
```

To include stopped or failed containers in the command output, add the **--all** option to the command:

```
$ sudo podman ps --all
```

To list container images, run the following command:

```
$ sudo podman images
```

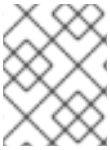
Inspecting container properties

To view the properties of a container or container images, use the **podman inspect** command. For example, to inspect the **keystone** container, run the following command:

```
$ sudo podman inspect keystone
```

Managing containers with Systemd services

Previous versions of OpenStack Platform managed containers with Docker and its daemon. In OpenStack Platform 16, the Systemd services interface manages the lifecycle of the containers. Each container is a service and you run Systemd commands to perform specific operations for each container.



NOTE

It is not recommended to use the Podman CLI to stop, start, and restart containers because Systemd applies a restart policy. Use Systemd service commands instead.

To check a container status, run the **systemctl status** command:

```
$ sudo systemctl status tripleo_keystone
● tripleo_keystone.service - keystone container
   Loaded: loaded (/etc/systemd/system/tripleo_keystone.service; enabled; vendor preset: disabled)
   Active: active (running) since Fri 2019-02-15 23:53:18 UTC; 2 days ago
 Main PID: 29012 (podman)
   CGroup: /system.slice/tripleo_keystone.service
           └─29012 /usr/bin/podman start -a keystone
```

To stop a container, run the **systemctl stop** command:

```
$ sudo systemctl stop tripleo_keystone
```

To start a container, run the **systemctl start** command:

```
$ sudo systemctl start tripleo_keystone
```

To restart a container, run the **systemctl restart** command:

```
$ sudo systemctl restart tripleo_keystone
```

Because no daemon monitors the containers status, Systemd automatically restarts most containers in these situations:

- Clean exit code or signal, such as running **podman stop** command.
- Unclean exit code, such as the podman container crashing after a start.
- Unclean signals.
- Timeout if the container takes more than 1m 30s to start.

For more information about Systemd services, see the **systemd.service** documentation.

NOTE

Any changes to the service configuration files within the container revert after restarting the container. This is because the container regenerates the service configuration based on files on the local file system of the node in **/var/lib/config-data/puppet-generated/**. For example, if you edit **/etc/keystone/keystone.conf** within the **keystone** container and restart the container, the container regenerates the configuration using **/var/lib/config-data/puppet-generated/keystone/etc/keystone/keystone.conf** on the local file system of the node, which overwrites any the changes that were made within the container before the restart.

Monitoring podman containers with Systemd timers

The Systemd timers interface manages container health checks. Each container has a timer that runs a service unit that executes health check scripts.

To list all OpenStack Platform containers timers, run the **systemctl list-timers** command and limit the output to lines containing **tripleo**:

```
$ sudo systemctl list-timers | grep tripleo
Mon 2019-02-18 20:18:30 UTC  1s left      Mon 2019-02-18 20:17:26 UTC  1min 2s ago
tripleo_nova_metadata_healthcheck.timer           tripleo_nova_metadata_healthcheck.service
Mon 2019-02-18 20:18:33 UTC  4s left      Mon 2019-02-18 20:17:03 UTC  1min 25s ago
tripleo_mistral_engine_healthcheck.timer        tripleo_mistral_engine_healthcheck.service
Mon 2019-02-18 20:18:34 UTC  5s left      Mon 2019-02-18 20:17:23 UTC  1min 5s ago
tripleo_keystone_healthcheck.timer          tripleo_keystone_healthcheck.service
Mon 2019-02-18 20:18:35 UTC  6s left      Mon 2019-02-18 20:17:13 UTC  1min 15s ago
tripleo_memcached_healthcheck.timer         tripleo_memcached_healthcheck.service
(...)
```

To check the status of a specific container timer, run the **systemctl status** command for the healthcheck service:

```
$ sudo systemctl status tripleo_keystone_healthcheck.service
● tripleo_keystone_healthcheck.service - keystone healthcheck
   Loaded: loaded (/etc/systemd/system/tripleo_keystone_healthcheck.service; disabled; vendor
preset: disabled)
   Active: inactive (dead) since Mon 2019-02-18 20:22:46 UTC; 22s ago
  Process: 115581 ExecStart=/usr/bin/podman exec keystone /openstack/healthcheck (code=exited,
status=0/SUCCESS)
 Main PID: 115581 (code=exited, status=0/SUCCESS)

Feb 18 20:22:46 undercloud.localdomain systemd[1]: Starting keystone healthcheck...
Feb 18 20:22:46 undercloud.localdomain podman[115581]: {"versions": {"values": [{"status": "stable",
"updated": "2019-01-22T00:00:00Z", "..."}]}]}}
Feb 18 20:22:46 undercloud.localdomain podman[115581]: 300 192.168.24.1:35357 0.012 seconds
Feb 18 20:22:46 undercloud.localdomain systemd[1]: Started keystone healthcheck.
```

To stop, start, restart, and show the status of a container timer, run the relevant **systemctl** command against the **.timer** Systemd resource. For example, to check the status of the **tripleo_keystone_healthcheck.timer** resource, run the following command:

```
$ sudo systemctl status tripleo_keystone_healthcheck.timer
● tripleo_keystone_healthcheck.timer - keystone container healthcheck
   Loaded: loaded (/etc/systemd/system/tripleo_keystone_healthcheck.timer; enabled; vendor preset:
disabled)
   Active: active (waiting) since Fri 2019-02-15 23:53:18 UTC; 2 days ago
```

If the healthcheck service is disabled but the timer for that service is present and enabled, it means that the check is currently timed out, but will be run according to timer. You can also start the check manually.

NOTE

The **podman ps** command does not show the container health status.

Checking container logs

OpenStack Platform 16 introduces a new logging directory **/var/log/containers/stdout** that contains the standard output (stdout) all of the containers, and standard errors (stderr) consolidated in one single file for each container.

Paunch and the **container-puppet.py** script configure podman containers to push their outputs to the **/var/log/containers/stdout** directory, which creates a collection of all logs, even for the deleted containers, such as **container-puppet-*** containers.

The host also applies log rotation to this directory, which prevents huge files and disk space issues.

In case a container is replaced, the new container outputs to the same log file, because **podman** uses the container name instead of container ID.

You can also check the logs for a containerized service with the **podman logs** command. For example, to view the logs for the **keystone** container, run the following command:

```
$ sudo podman logs keystone
```

Accessing containers

To enter the shell for a containerized service, use the **podman exec** command to launch **/bin/bash**. For example, to enter the shell for the **keystone** container, run the following command:

```
$ sudo podman exec -it keystone /bin/bash
```

To enter the shell for the **keystone** container as the root user, run the following command:

```
$ sudo podman exec --user 0 -it <NAME OR ID> /bin/bash
```

To exit the container, run the following command:

```
# exit
```

5.2. TROUBLESHOOTING CONTAINERIZED SERVICES

If a containerized service fails during or after overcloud deployment, use the following recommendations to determine the root cause for the failure:

NOTE

Before running these commands, check that you are logged into an overcloud node and not running these commands on the undercloud.

Checking the container logs

Each container retains standard output from its main process. This output acts as a log to help determine what actually occurs during a container run. For example, to view the log for the **keystone** container, use the following command:

```
$ sudo podman logs keystone
```

In most cases, this log provides the cause of a container's failure.

Inspecting the container

In some situations, you might need to verify information about a container. For example, use the following command to view **keystone** container data:

```
$ sudo podman inspect keystone
```

This provides a JSON object containing low-level configuration data. You can pipe the output to the **jq** command to parse specific data. For example, to view the container mounts for the **keystone** container, run the following command:

```
$ sudo podman inspect keystone | jq .[0].Mounts
```

You can also use the **--format** option to parse data to a single line, which is useful for running commands against sets of container data. For example, to recreate the options used to run the **keystone** container, use the following **inspect** command with the **--format** option:

```
$ sudo podman inspect --format='{{range .Config.Env}} -e "{{.}}" {{end}} {{range .Mounts}} -v
{{.Source}}:{{.Destination}}{{if .Mode}}:{{.Mode}}{{end}}{{end}} -ti {{.Config.Image}}' keystone
```

NOTE

The **--format** option uses Go syntax to create queries.

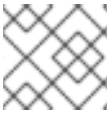
Use these options in conjunction with the **podman run** command to recreate the container for troubleshooting purposes:

```
$ OPTIONS=$( sudo podman inspect --format='{{range .Config.Env}} -e "{{.}}" {{end}} {{range
.Mounts}} -v {{.Source}}:{{.Destination}}{{if .Mode}}:{{.Mode}}{{end}}{{end}} -ti {{.Config.Image}}'
keystone )
$ sudo podman run --rm $OPTIONS /bin/bash
```

Running commands in the container

In some cases, you might need to obtain information from within a container through a specific Bash command. In this situation, use the following **podman** command to execute commands within a running container. For example, to run a command in the **keystone** container:

```
$ sudo podman exec -ti keystone <COMMAND>
```



NOTE

The **-ti** options run the command through an interactive pseudoterminal.

Replace **<COMMAND>** with your desired command. For example, each container has a health check script to verify the service connection. You can run the health check script for **keystone** with the following command:

```
$ sudo podman exec -ti keystone /openstack/healthcheck
```

To access the container's shell, run **podman exec** using **/bin/bash** as the command:

```
$ sudo podman exec -ti keystone /bin/bash
```

Exporting a container

When a container fails, you might need to investigate the full contents of the file. In this case, you can export the full file system of a container as a **tar** archive. For example, to export the **keystone** container's file system, run the following command:

```
$ sudo podman export keystone -o keystone.tar
```

This command create the **keystone.tar** archive, which you can extract and explore.

CHAPTER 6. COMPARING SYSTEMD SERVICES TO CONTAINERIZED SERVICES

This chapter provides some reference material to show how containerized services differ from Systemd services.

6.1. SYSTEMD SERVICES AND CONTAINERIZED SERVICES

The following table shows the correlation between Systemd-based services and the **podman** containers controlled with the Systemd services.

Component	Systemd service	Containers
OpenStack Image Storage (glance)	**tripleo_glance_api.service**	**glance_api**
HAProxy	**tripleo_haproxy.service**	**haproxy**
OpenStack Orchestration (heat)	**tripleo_heat_api.service** **tripleo_heat_api_cfn.service** **tripleo_heat_api_cron.service** **tripleo_heat_engine.service**	**heat_api** **heat_api_cfn** **heat_api_cron** **heat_engine**
OpenStack Bare Metal (ironic)	**tripleo_ironic_api.service** **tripleo_ironic_conductor.service** **tripleo_ironic_inspector.service** **tripleo_ironic_inspector_dnsmasq.service** **tripleo_ironic_neutron_agent.service** **tripleo_ironic_pxe_http.service** **tripleo_ironic_pxe_tftp.service** **tripleo_iscsid.service**	**ironic_api** **ironic_conductor** **ironic_inspector** **ironic_inspector_dnsmasq** **ironic_neutron_agent** **ironic_pxe_http** **ironic_pxe_tftp** **iscsid**
Keepalived	**tripleo_keepalived.service**	**keepalived**
OpenStack Identity (keystone)	**tripleo_keystone.service** **tripleo_keystone_cron.service**	**keystone** **keystone_cron**
Logrotate	**tripleo_logrotate_crond.service**	**logrotate_crond**
Memcached	**tripleo_memcached.service**	**memcached**

Component	Systemd service	Containers
OpenStack Workflow (mistral)	**tripleo_mistral_api.service** **tripleo_mistral_engine.service** **tripleo_mistral_event_engine.service** **tripleo_mistral_executor.service**	**mistral_api** **mistral_engine** **mistral_event_engine** **mistral_executor**
MySQL	**tripleo_mysql.service**	**mysql**
OpenStack Networking (neutron)	**tripleo_neutron_api.service** **tripleo_neutron_dhcp.service** **tripleo_neutron_l3_agent.service** **tripleo_neutron_ovs_agent.service**	**neutron_api** **neutron_dhcp** **neutron_l3_agent** **neutron_ovs_agent**
OpenStack Compute (nova)	**tripleo_nova_api.service** **tripleo_nova_api_cron.service** **tripleo_nova_compute.service** **tripleo_nova_conductor.service** **tripleo_nova_metadata.service** **tripleo_nova_placement.service** **tripleo_nova_scheduler.service**	**nova_api** **nova_api_cron** **nova_compute** **nova_conductor** **nova_metadata** **nova_placement** **nova_scheduler**
RabbitMQ	**tripleo_rabbitmq.service**	**rabbitmq**
OpenStack Object Storage (swift)	**tripleo_swift_account_reaper.service** **tripleo_swift_account_server.service** **tripleo_swift_container_server.service** **tripleo_swift_container_updater.service** **tripleo_swift_object_expirer.service** **tripleo_swift_object_server.service** **tripleo_swift_object_updater.service** **tripleo_swift_proxy.service** **tripleo_swift_rsync.service**	**swift_account_reaper** **swift_account_server** **swift_container_server** **swift_container_updater** **swift_object_expirer** **swift_object_server** **swift_object_updater** **swift_proxy** **swift_rsync**

Component	Systemd service	Containers
OpenStack Messaging (zaqar)	**tripleo_zaqar.service** **tripleo_zaqar_websocket.service**	**zaqar** **zaqar_websocket**

6.2. SYSTEMD LOG LOCATIONS VS CONTAINERIZED LOG LOCATIONS

The following table shows Systemd-based OpenStack logs and their equivalents for containers. All container-based log locations are available on the physical host and are mounted to the container.

OpenStack service	Systemd service logs	Container logs
aodh	**/var/log/aodh/**	**/var/log/containers/aodh/** **/var/log/containers/httpd/aodh-api/**
ceilometer	**/var/log/ceilometer/**	**/var/log/containers/ceilometer/**
cinder	**/var/log/cinder/**	**/var/log/containers/cinder/** **/var/log/containers/httpd/cinder-api/**
glance	**/var/log/glance/**	**/var/log/containers/glance/**
gnocchi	**/var/log/gnocchi/**	**/var/log/containers/gnocchi/** **/var/log/containers/httpd/gnocchi-api/**
heat	**/var/log/heat/**	**/var/log/containers/heat/** **/var/log/containers/httpd/heat-api/** **/var/log/containers/httpd/heat-api-cfn/**
horizon	**/var/log/horizon/**	**/var/log/containers/horizon/** **/var/log/containers/httpd/horizon/**

OpenStack service	Systemd service logs	Container logs
keystone	**/var/log/keystone/**	**/var/log/containers/keystone** **/var/log/containers/httpd/keystone/**
databases	**/var/log/mariadb/** **/var/log/mongodb/** **/var/log/mysqld.log**	**/var/log/containers/mysql/**
neutron	**/var/log/neutron/**	**/var/log/containers/neutron/** **/var/log/containers/httpd/neutron-api/**
nova	**/var/log/nova/**	**/var/log/containers/nova/** **/var/log/containers/httpd/nova-api/** **/var/log/containers/httpd/placement/**
panko		**/var/log/containers/panko/** **/var/log/containers/httpd/panko-api/**
rabbitmq	**/var/log/rabbitmq/**	**/var/log/containers/rabbitmq/**
redis	**/var/log/redis/**	**/var/log/containers/redis/**
swift	**/var/log/swift/**	**/var/log/containers/swift/**

6.3. SYSTEMD CONFIGURATION VS CONTAINERIZED CONFIGURATION

The following table shows Systemd-based OpenStack configuration and their equivalents for containers. All container-based configuration locations are available on the physical host, are mounted to the container, and are merged (via **kolla**) into the configuration within each respective container.

OpenStack service	Systemd service configuration	Container configuration
aodh	**/etc/aodh/**	**/var/lib/config-data/puppet-generated/aodh/**

OpenStack service	Systemd service configuration	Container configuration
ceilometer	**/etc/ceilometer/**	**/var/lib/config-data/puppet-generated/ceilometer/etc/ceilometer/**
cinder	**/etc/cinder/**	**/var/lib/config-data/puppet-generated/cinder/etc/cinder/**
glance	**/etc/glance/**	**/var/lib/config-data/puppet-generated/glance_api/etc/glance/**
gnocchi	**/etc/gnocchi/**	**/var/lib/config-data/puppet-generated/gnocchi/etc/gnocchi/**
haproxy	**/etc/haproxy/**	**/var/lib/config-data/puppet-generated/haproxy/etc/haproxy/**
heat	**/etc/heat/**	**/var/lib/config-data/puppet-generated/heat/etc/heat/** **/var/lib/config-data/puppet-generated/heat_api/etc/heat/** **/var/lib/config-data/puppet-generated/heat_api_cfn/etc/heat/**
horizon	**/etc/openstack-dashboard/**	**/var/lib/config-data/puppet-generated/horizon/etc/openstack-dashboard/**
keystone	**/etc/keystone/**	**/var/lib/config-data/puppet-generated/keystone/etc/keystone/**
databases	**/etc/my.cnf.d/** **/etc/my.cnf**	**/var/lib/config-data/puppet-generated/mysql/etc/my.cnf.d/**
neutron	**/etc/neutron/**	**/var/lib/config-data/puppet-generated/neutron/etc/neutron/**

OpenStack service	Systemd service configuration	Container configuration
nova	**/etc/nova/**	**/var/lib/config-data/puppet-generated/nova/etc/nova/** **/var/lib/config-data/puppet-generated/etc/placement/**
panko		**/var/lib/config-data/puppet-generated/panko/etc/panko**
rabbitmq	**/etc/rabbitmq/**	**/var/lib/config-data/puppet-generated/rabbitmq/etc/rabbitmq/**
redis	**/etc/redis/** **/etc/redis.conf**	**/var/lib/config-data/puppet-generated/redis/etc/redis/** **/var/lib/config-data/puppet-generated/redis/etc/redis.conf**
swift	**/etc/swift/**	**/var/lib/config-data/puppet-generated/swift/etc/swift/** **/var/lib/config-data/puppet-generated/swift_ringbuilder/etc/swift/**