OS Migrate Documentation

Table of Contents

1. Welcome to OS Migrate

OS Migrate provides a framework and toolsuite for exporting and importing resources between two clouds. It's a collection of Ansible playbooks that provide the basic functionality, but may not fit each use case out of the box. You can craft custom playbooks using the OS Migrate collection pieces (roles and modules) as building blocks.

At present OS Migrate supports migration from VMware clouds to OpenStack, and OpenStack to OpenStack.

OS Migrate strictly uses the official OpenStack API and does not utilize direct database access or other methods to export or import data. The Ansible playbooks contained in OS Migrate are idempotent. If a command fails, you can retry with the same command.

2. Community

The source code of OS Migrate is hosted on GitHub.

For issue reports please use the GitHub OS Migrate issue tracker.

To get help, feel free to also create an issue on GitHub with your question.

If you want to contribute to the project (code, docs, …), please refer to the developer docs.

3. Contents

4. OS Migrate Operator Guide

This guide provides comprehensive information for operators planning and executing OpenStack cloud migrations using OS Migrate.

4.1. Overview

4.2. os-migrate overview

OS-Migrate is a toolsuite designed to assist in the process of moving cloud workloads from one environment to another. Please read the entire workflow notes in order to be sufficiently prepared for this endeavour.

TO BE FLESHED OUT

4.3. Planning and Setup

4.4. OS Migrate Capacity Planning

4.4.1. Migrator Host Requirements

Overview

The migrator host is the machine that executes OS Migrate Ansible playbooks and serves as the orchestration point for OpenStack cloud migrations. It can be deployed as:

- A source cloud instance
- A third-party machine (physical/virtual)
- An Ansible Execution Environment (container)

System Requirements

Operating System

- Recommended: CentOS Stream 10 or RHEL 9.5+
- Alternative: Any Linux distribution with virtio-win package ≥1.40 (for workload migrations)
- Container environments supported for Ansible Execution Environments

Software Dependencies

Core Requirements

- Python: 3.7+ (badges show 3.7+, container uses 3.12)
- Ansible: 2.9+ minimum (current: 11.7.0)

Python Packages

```
ansible==11.7.0
ansible-lint==25.6.1
openstacksdk==1.5.1
python-openstackclient==6.3.0
passlib==1.7.4
PyYAML==6.0.2
```

Ansible Collections

```
community.crypto: ">=1.4.0"
community.general: "<5.0.0"
openstack.cloud: ">=2.3.0,<2.4.0"
```

Storage Requirements

- Sufficient disk space in os_migrate_data_dir (default: /home/migrator/os-migrate-data)
- Separate data directories required for each source project migration
- Additional space for volume conversion during workload migrations

Network Access

- Outbound: HTTPS access to both source and destination OpenStack APIs
- Inbound: SSH access if used as remote conversion host
- Bandwidth: Adequate for volume/image data transfer during migrations

Configuration Requirements

Authentication

- Valid OpenStack credentials for both source and destination clouds
- clouds.yaml configuration or environment variables
- Support for Keystone v2/v3 authentication types

Inventory Configuration

```
migrator:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python_interpreter: "{{ ansible_playbook_python }}"
```

Required Variables

- os_migrate_src_auth: Source cloud authentication
- os_migrate_dst_auth: Destination cloud authentication
- os_migrate_data_dir: Migration data storage location
- Region and API version configurations as needed

Deployment Options

1. Source Cloud Instance

- Pros: Close network proximity to source resources
- Cons: Resource consumption on source cloud
- Use case: When destination cloud has limited external access

2. Third-Party Machine

- Pros: Independent resource allocation, flexible placement

- Cons: Network latency considerations
- Use case: Large migrations, dedicated migration infrastructure

3. Ansible Execution Environment

- Pros: Consistent, portable, containerized execution
- Cons: Container orchestration complexity
- Use case: CI/CD pipelines, automated migrations

Special Considerations

Workload Migration

- Additional conversion host may be required for instance migrations
- SSH key access to conversion hosts
- Volume attachment/detachment capabilities

Multi-Project Migrations

- Separate data directories for each source project
- Proper isolation of migration data
- Sequential or parallel execution planning

Security

- No direct database access (API-only operations)
- Secure credential management
- Network security between clouds

4.5. Conversion Host Guide

The conversion host role is a critical component of OS Migrate for migrating OpenStack workloads (instances) and their associated volumes between OpenStack clouds. It deploys dedicated compute instances that act as intermediary hosts for data transfer and conversion operations during workload migration.

4.5.1. Purpose

Conversion hosts serve as temporary staging instances that:

- **Data Transfer**: Facilitate the transfer of workload disk data between source and destination clouds
- **Format Conversion**: Handle disk format conversions if needed between different storage backends
- **Network Bridging**: Provide network connectivity between source and destination

environments
- **Migration Orchestration**: Execute the complex multi-step process of workload migration

4.5.2. Architecture

The conversion host system consists of two main Ansible roles:

`conversion_host` Role

Location: `/roles/conversion_host/`

Purpose: Deploys the infrastructure and compute instances needed for conversion hosts.

Key Responsibilities:

- Creates networking infrastructure (networks, subnets, routers, security groups)
- Generates SSH keypairs for secure communication
- Deploys conversion host instances in both source and destination clouds
- Configures floating IPs for external connectivity
- Sets up security group rules for SSH and ICMP access

`conversion_host_content` Role

Location: `/roles/conversion_host_content/`

Purpose: Installs and configures software packages on the conversion hosts.

Key Responsibilities:

- Installs OS-specific packages (supports CentOS and RHEL)
- Configures RHEL subscription management if needed
- Sets up SSH keys for inter-host communication
- Enables password access if required
- Runs pre/post installation hooks

4.5.3. How It Works

Deployment Process

1. **Infrastructure Setup** (`conversion_host` role):

   ```
   # Network Infrastructure
   - Creates conversion network (default: os_migrate_conv)
   - Creates subnet with CIDR 192.168.10.0/24
   - Creates router connected to external network
   - Creates security group with SSH/ICMP rules
   ```

2. **Instance Deployment**:

   ```
   # Source Cloud Instance
   os_migrate_src_conversion_host_name: os_migrate_conv_src

   # Destination Cloud Instance
   os_migrate_dst_conversion_host_name: os_migrate_conv_dst
   ```

3. **SSH Key Configuration**:
   - Generates keypair for conversion host access
   - Configures authorized keys on source host
   - Installs private key on destination host for src→dst communication
4. **Software Installation** (`conversion_host_content` role):
   - Installs required packages based on OS distribution
   - Configures any RHEL subscriptions
   - Sets up conversion tools and dependencies

Migration Workflow Integration

During workload migration, conversion hosts are used in the following workflow:

1. **Volume Export**: Source conversion host exports volumes from source cloud storage
2. **Data Transfer**: Volumes are transferred from source to destination conversion host
3. **Volume Import**: Destination conversion host imports volumes to destination storage
4. **Instance Creation**: New instance is created using imported volumes

Network Architecture

```
Source Cloud                 Destination Cloud
+------------------+        +------------------+
| Conversion Host  |        | Conversion Host  |
| (os_migrate_conv)|   SSH  | (os_migrate_conv)|
| 192.168.10.x     |<------>| 192.168.10.x     |
+------------------+        +------------------+
         |                           |
    [Floating IP]              [Floating IP]
         |                           |
    [External Net]             [External Net]
```

4.5.4. VMware as source

In the case of VMware vCenter or ESXi as the source cloud, there is no conversion host in the source cloud (VMware), but only in the OpenStack destination cloud. The conversion host in the destination cloud will handle all connections and data transfers from VMware to OpenStack. This is done first via the VMware APIs on port 443 to set up and collect the required information, and then using NBD on port 902 during the data transfer. Ensure that the ports are correctly configured at the firewall level and that the FQDN of the VMware target is functional.

The diagram below explains how OS-Migrate uses the conversion host to handle data transfer. First, the migration process connects to the vCenter or ESXi host via HTTPS to gather information such as the full disk path in the datastore, take a snapshot of the running VMs, retrieve the change block tracking ID (CBT), manage the power state of the VMs, and finally clean up the snapshots after the migration succeeds. In the second phase, the migration process starts an NBD (Network Block Device) server to connect to the VMware datastore and read the disk from the previously taken snapshot. Port 902 is used for this connection, and once established, the data transfer occurs via the conversion host to a Cinder volume created by OS-Migrate. It is important to ensure that all ports and host resolutions are correctly configured on the conversion host side. Once the data transfer is complete, the migration process converts the filesystem to run under KVM. At this point, the migration is finished.

```
Source VMware                       Destination Cloud
+------------------+                +------------------+
|                  |    902 (nbd)   |                  |
| VCenter/ESXi     |<-------------->| Conversion Host  |
| IP/FQDN          |   443 (https)  | (os_migrate_conv)|
| Guest Snapshots  |<-------------->| 192.168.10.x     |
+------------------+                +------------------+
         |                                   |
                                       [Floating IP]
         |                                   |
                                       [External Net]
```

4.5.5. Configuration

Required Variables

```
# Must be defined
os_migrate_conversion_flavor_name: m1.large
os_migrate_conversion_external_network_name: public
```

Network Configuration

```
# Network settings (with defaults)
os_migrate_conversion_net_name: os_migrate_conv
os_migrate_conversion_subnet_name: os_migrate_conv
os_migrate_conversion_subnet_cidr: 192.168.10.0/24
os_migrate_conversion_subnet_alloc_start: 192.168.10.10
os_migrate_conversion_subnet_alloc_end: 192.168.10.99
os_migrate_conversion_router_name: os_migrate_conv
os_migrate_conversion_router_ip: 192.168.10.1
```

Host Configuration

```
# Instance settings
os_migrate_src_conversion_host_name: os_migrate_conv_src
os_migrate_dst_conversion_host_name: os_migrate_conv_dst
os_migrate_conversion_image_name: os_migrate_conv
os_migrate_conversion_host_ssh_user: cloud-user

# Boot from volume options
os_migrate_src_conversion_host_boot_from_volume: false
os_migrate_dst_conversion_host_boot_from_volume: false
os_migrate_src_conversion_host_volume_size: 20
os_migrate_dst_conversion_host_volume_size: 20
```

Security Configuration

```
# SSH and security
os_migrate_conversion_keypair_name: os_migrate_conv
os_migrate_conversion_keypair_private_path: "{{ os_migrate_data_dir
}}/conversion/ssh.key"
os_migrate_conversion_secgroup_name: os_migrate_conv

# Password access (disabled by default)
os_migrate_conversion_host_ssh_user_enable_password_access: false
```

Management Options

```
# Infrastructure management
os_migrate_conversion_manage_network: true
os_migrate_conversion_manage_fip: true
os_migrate_conversion_delete_fip: true

# Content installation
os_migrate_conversion_host_content_install: true

# Deployment control
os_migrate_deploy_src_conversion_host: true
os_migrate_deploy_dst_conversion_host: true
os_migrate_link_conversion_hosts: true
os_migrate_reboot_conversion_hosts: false
```

4.5.6. Usage

Deploy Conversion Hosts

Use the provided playbook to deploy conversion hosts:

```
ansible-playbook os_migrate.os_migrate.deploy_conversion_hosts
```

This playbook will:

1. Deploy source conversion host infrastructure and instance
2. Deploy destination conversion host infrastructure and instance
3. Configure SSH linking between hosts
4. Install required software packages
5. Perform health checks

Delete Conversion Hosts

Clean up conversion hosts and their infrastructure:

```
ansible-playbook os_migrate.os_migrate.delete_conversion_hosts
```

Manual Role Usage

You can also use the roles directly for more control:

```
# Deploy source conversion host
- name: Deploy source conversion host
  include_role:
    name: os_migrate.os_migrate.conversion_host
  vars:
    os_migrate_conversion_cloud: src
    os_migrate_conversion_host_name: "{{ os_migrate_src_conversion_host_name }}"
    # ... other source-specific variables

# Deploy destination conversion host
- name: Deploy destination conversion host
  include_role:
    name: os_migrate.os_migrate.conversion_host
  vars:
    os_migrate_conversion_cloud: dst
    os_migrate_conversion_host_name: "{{ os_migrate_dst_conversion_host_name }}"
    # ... other destination-specific variables
```

4.5.7. Integration with Workload Migration

Conversion hosts are automatically used during workload migration when:

1. **Workload Export/Import**: The `import_workloads` role checks conversion host status
2. **Data Copy Operations**: Volume data is transferred via conversion hosts

3. **Health Checks**: Ensures conversion hosts are active and reachable before migration

The `os_conversion_host_info` module provides runtime information about conversion hosts:

```
- name: Get conversion host info
  os_migrate.os_migrate.os_conversion_host_info:
    cloud: src
    server: "{{ os_migrate_src_conversion_host_name }}"
  register: conversion_host_info
```

4.5.8. Prerequisites

Cloud Requirements

- **Flavor**: Adequate flavor for conversion operations (recommended: >= 2 vCPU, 4GB RAM)
- **Image**: Compatible base image (CentOS/RHEL with cloud-init)
- **Network**: External network for floating IP assignment
- **Quotas**: Sufficient quota for additional instances, networks, and floating IPs

Permissions

The deployment requires OpenStack permissions for:

- Instance creation/deletion
- Network resource management (networks, subnets, routers)
- Security group management
- Keypair management
- Floating IP allocation

4.5.9. Troubleshooting

Common Issues

1. Conversion Host Not Reachable

```
# Check conversion host status
- os_conversion_host_info module will fail if host is not ACTIVE
- Verify floating IP assignment
- Check security group rules allow SSH (port 22)
```

2. Network Connectivity Issues

```
# Verify network configuration
- External network name is correct
- Router has gateway set to external network
```

```
- DNS nameservers are accessible (default: 8.8.8.8)
```

3. SSH Key Problems

```
# Key file permissions
- Private key must have 0600 permissions
- Public key must be accessible to Ansible
- Verify keypair exists in OpenStack
```

4. Package Installation Failures

```
# RHEL subscription issues
- Check RHEL subscription configuration
- Verify repository access
- Review pre/post content hooks
```

Debugging Steps

1. **Check Conversion Host Status**:

   ```yaml
   - name: Debug conversion host
     os_migrate.os_migrate.os_conversion_host_info:
       cloud: src
       server: "{{ os_migrate_src_conversion_host_name }}"
     register: debug_info

   - debug: var=debug_info
   ```

2. **Verify Network Connectivity**:

   ```yaml
   - name: Test SSH connectivity
     wait_for:
       port: 22
       host: "{{ conversion_host_ip }}"
       timeout: 60
   ```

3. **Check Infrastructure**:

   ```bash
   # Verify OpenStack resources exist
   openstack server list --name os_migrate_conv
   openstack network list --name os_migrate_conv
   openstack security group list --name os_migrate_conv
   ```

4.5.10. Best Practices

Security

- Use dedicated keypairs for conversion hosts
- Limit security group rules to necessary ports only
- Consider using specific subnets for conversion traffic
- Disable password authentication unless specifically required

Performance

- Choose appropriate flavors with sufficient CPU and memory
- Consider boot-from-volume for larger disk operations
- Use local storage-optimized flavors when available
- Monitor network bandwidth during large migrations

Cleanup

- Always clean up conversion hosts after migration
- Set `os_migrate_conversion_delete_fip: true` to clean floating IPs
- Use the delete playbook to ensure complete cleanup
- Monitor for orphaned resources after deletion

Testing

- Test conversion host deployment in development environment first
- Verify connectivity between source and destination hosts
- Test package installation and configuration
- Validate migration workflow with small test instances

4.5.11. Related Components

- `import_workloads` **role**: Uses conversion hosts for actual migration operations
- `os_conversion_host_info` **module**: Provides runtime host information
- **Volume migration modules**: Transfer data via conversion hosts
- **Workload migration utilities**: Coordinate conversion host operations

4.5.12. Examples

Basic Deployment

```
# Minimal configuration for conversion host deployment
os_migrate_conversion_flavor_name: m1.large
```

```
os_migrate_conversion_external_network_name: public
os_migrate_src_conversion_image_name: centos-stream-9
os_migrate_dst_conversion_image_name: centos-stream-9
```

Advanced Configuration

```
# Advanced configuration with custom networks
os_migrate_conversion_flavor_name: m1.xlarge
os_migrate_conversion_external_network_name: external
os_migrate_src_conversion_net_name: migration_src_net
os_migrate_dst_conversion_net_name: migration_dst_net
os_migrate_conversion_subnet_cidr: 10.10.10.0/24
os_migrate_src_conversion_host_boot_from_volume: true
os_migrate_dst_conversion_host_boot_from_volume: true
os_migrate_src_conversion_host_volume_size: 50
os_migrate_dst_conversion_host_volume_size: 50
```

4.6. VMware to OpenStack OS Migrate Performance Expectations

This document outlines some performance expectations for VMware to OpenStack transfers via the os-migrate tooling. For the purposes of the these metrics, we are considering the following hardware for a Conversion Host:

- 6 vCPU
- 8GB RAM
- 20GB disk unless otherwise stated

All of the below metrics apply to VMware to OpenStack via the Ansible Automation Platform

Table 1

Time of execution to completion including instance boot & ping for RHEL machines in our lab.

VMs	Conversion Hosts	Threads	Time
1	1	1	2 minutes
20	1	1	31 minutes
20	1	2	15 minutes
30	1	10	8 minutes
100	1	10	20 minutes

Note: The average time for 1 virtual machine is around 2 minutes. The migration can be parallelized on the same conversion host (threads) or on multiple conversion hosts.

Table 2

Time for RHEL machines with a larger disk for a full migration run with and without CBT[1] options:

Disk size	CBT enabled	Time total	Cutover time	Expected downtime
100 GB	yes	8 minutes	2 minutes	2 minutes
100 GB	no	7 minutes		7 minutes
200 GB	yes	10 minutes 30 secondes	2 minutes	2 minutes
200 GB	no	9 minutes 30 secondes		9 minutes 30 secondes
300 GB	yes	17 minutes	2 minutes	2 minutes
300 GB	no	15 minutes		15 minutes
1 TB	yes	39 minutes	2 minutes	2 minutes
1 TB	no	35 minutes		35 minutes

Note 1: The disks are 99% full of random data during the test.

Note 2: The CBT option takes a bit more time overall but the downtime is actually lower due to the smaller data cutover time.

Table 3

Example of a Migration plan for 55 VMs with 200 GB of disk full. *(estimation)*

Disk size	Conversion Hosts	Threads	CBT enabled	Migration time	Sync time
200 GB	1	1	yes	115 minutes	
200 GB	1	5	no	105 minutes	
200 GB	1	5	yes	22 minutes	110 minutes

For this example, the best plan is to parallelize the migration on a single Conversion Host to 5 threads and use the CBT option to pre-migrate the volume data; this will shortern the cutover time (2 minutes in our lab environment), reducing the downtime overall.

Table 4

Migration of 1500 VMs: Time for RHEL machines in our lab with a disk capacity of 20GB, with a full migration run which comprises of migration of data, conversion and instance creation, boot time, and ability to ping.

VMs	Conversion Hosts	Threads	Time

250	2	6	60 minutes (without the preparation steps)
1000	2	6	5 hours
1500	2	6	

Note 1: The conversion flavor was set to: **12GB of RAM and 6 vcpus** which allow OS-Migrate to comfortably run 6 migrations in parallel on the same host. There is space to execute more parallel migrations with this configuration but it’s a best practice to let 2GB of RAM and 1 CPU for each migration.

Conversion host requirements and recommendations

Recommended guidelines:

- **For 1 migration allocate 2GB of RAM and 1 vCPU on your Conversion Host**
- RHEL 9.5 or CentOS 10 to benefit from the latest drivers (virtio-win) to convert the recent Windows distributions.
- Fedora (38 and +) if you want to convert btrfs file system

How to proceed with a large workload

It’s important to first focus on sizing your target OpenStack environment adequately. It will of course receive large numbers of instances, ports, volumes, floating ips and all the OpenStack resources that the instances will require. So set the OpenStack quotas with the correct values.

Second you need to split your workload regarding the number of conversion hosts you will create.

For example, with a workload of 1000 VMs, you can use two conversion hosts which will run the migration in parallel and execute 5 or 6 migrations simultaneously on each conversion host.

If you followed the requirements for the conversion host above, the migration time will be linear:

If 1 virtual machine takes 5 minutes to be migrated then the 1000 Vms will take:

- 5*1000/12 (where 12 is 6 parallel migrations x 2 conversion hosts): 6 hours and 56 minutes

The more you divide your workload, the faster you will move it to OpenStack.

In the example above, if we decide to add 2 more conversion hosts then:

- 5*1000/24 = 3h and 28 minutes.

Finally, some known issues we have seen may happen during a large-scale migration.

- First, when the conversion hosts are too solicited. For example, a single conversion host which has performed more than 5 000 of migrations in a very short time may have some issues with the device mount mechanism in the OS itself.

When this behavior appears, sometimes just a reboot might help to clean the /dev/ devices.

- Another issue may appear on the Vmware side with the message "snapshot hierarchy is too deep" because OS-Migrate is working with Snapshot. If this error appears, then clean the guest snapshots hierarchy and re-run the migration.

Note 1 - Make sure all the OpenStack services are configured to support a large amount of requests. OS-Migrate is driven by Ansible but the core of the migration is a binary which does not consume a lot of resources. So the more you use the binary, the more the OpenStack Api will receive requests. For example Rabbitmq, Galera and also Nova or Cinder will be impacted.

4.7. Installation

4.8. Installation from Galaxy (recommended)

This document describes the recommended method of installing OS Migrate, from Ansible Galaxy. Alternatively, you can install from source.

4.8.1. Prerequisites

- Ansible 2.9 or newer.
- Ansible must run using Python 3. (When using Ansible from RHEL packages, this means running on RHEL 8 or newer.)
- Additional package requirements from Ansible modules: `iputils` `python3-openstackclient` `python3-openstacksdk`

4.8.2. Using virtualenv for prerequisites

If your distribution doesn’t ship the required dependency versions, you can use virtualenv, e.g.:

```
python3 -m venv $HOME/os_migrate_venv
source $HOME/os_migrate_venv/bin/activate
python3 -m pip install --upgrade 'openstacksdk'
python3 -m pip install --upgrade 'ansible-core'
```

4.8.3. Installation of OS Migrate collection

To install latest release:

```
ansible-galaxy collection install os_migrate.os_migrate
```

To install a particular release:

```
ansible-galaxy collection install os_migrate.os_migrate:<VERSION>
```

You can find available releases at OS Migrate Galaxy page.

4.9. Installation from source (when customizing)

Install prerequisites as documented in install from Galaxy. Then install OS Migrate from source:

```
git clone https://github.com/os-migrate/os-migrate
cd os-migrate
# > Here you can checkout a specific commit/branch if desired <

make toolbox-build
./toolbox/run make

pushd releases
ansible-galaxy collection install --force os_migrate-os_migrate-latest.tar.gz
popd
```

4.10. Ansible Execution Environment (AEE) Images

os-migrate and vmware-migration-kit provide containerized Ansible Execution Environment (AEE) images that encapsulate all necessary dependencies for running migration playbooks in a consistent, isolated environment.

4.10.1. Overview

Ansible Execution Environments are container images that provide a standardized runtime environment for Ansible automation. They include:

- Ansible Core and Ansible Runner
- Python runtime and required packages
- os-migrate and vmware-migration-kit collections
- All necessary dependencies and tools

This approach ensures consistent behavior across different environments and simplifies deployment and maintenance.

4.10.2. Available AEE Images

os-migrate AEE

The os-migrate AEE image contains:

- Ansible Core
- os-migrate Ansible collection
- OpenStack SDK and related Python packages
- All required dependencies for OpenStack resource migration

vmware-migration-kit AEE

The vmware-migration-kit AEE image contains:

- Ansible Core
- vmware-migration-kit Ansible collection
- VMware SDK and related Python packages
- All required dependencies for VMware to OpenStack migration

4.10.3. Building AEE Images

Prerequisites

Before building AEE images, ensure you have the following tools installed:

- `ansible-builder` - Tool for building execution environments
- `podman` or `docker` - Container runtime
- `git` - Version control system
- `python3` - Python runtime (version 3.8 or higher)

Setting Up a Virtual Environment

It’s recommended to use a Python virtual environment to isolate dependencies and avoid conflicts with system packages.

Create and activate a virtual environment:

```
# Create virtual environment
python3 -m venv .venv

# Activate virtual environment (Linux/macOS)
source .venv/bin/activate

# Activate virtual environment (Windows)
.venv\Scripts\activate
```

Installing Dependencies

Install the required dependencies using the project-specific requirements files:

For os-migrate:

```
# Clone the repository
git clone https://github.com/os-migrate/os-migrate.git
cd os-migrate

# Create and activate virtual environment
```

```
python3 -m venv .venv
source .venv/bin/activate

# Install build dependencies
pip install -r requirements-build.txt
```

For vmware-migration-kit:

```
# Clone the repository
git clone https://github.com/os-migrate/vmware-migration-kit.git
cd vmware-migration-kit

# Create and activate virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install build dependencies
pip install -r requirements-build.txt
```

Requirements Files

Both repositories provide `requirements-build.txt` files that contain all necessary dependencies for building AEE images:

- **os-migrate requirements**: https://github.com/os-migrate/os-migrate/blob/main/requirements-build.txt
- **vmware-migration-kit requirements**: https://github.com/os-migrate/vmware-migration-kit/blob/main/requirements-build.txt

These files include: * `ansible-builder` - Core tool for building execution environments * `ansible-core` - Ansible runtime * `ansible-runner` - Execution environment runner * Additional Python packages required for the build process

Collection Requirements in AEE

The AEE images use `requirements.yml` files to specify which Ansible collections to install. The collection installation method depends on the build context:

For main branch builds (development):

Install collections directly from Git repositories using the main branch:

```
# requirements.yml for main branch builds
collections:
  - name: https://github.com/os-migrate/os-migrate.git
    type: git
    version: main
  - name: https://github.com/os-migrate/vmware-migration-kit.git
    type: git
```

```
    version: main
```

For stable/tagged builds (production):

Install collections from Ansible Galaxy using specific version tags:

```
# requirements.yml for stable/tagged builds
collections:
  - name: os_migrate.os_migrate
    version: "1.0.1"
  - name: os_migrate.vmware_migration_kit
    version: "2.0.4"
```

Benefits of this approach:

- **Main branch builds**: Always get the latest development code with latest features and fixes
- **Stable builds**: Use tested, released versions for production stability
- **Version consistency**: AEE image tags match the collection versions they contain
- **Reproducible builds**: Same collection versions produce identical AEE images

Alternative Installation Methods

If you prefer not to use virtual environments, you can install ansible-builder globally:

```
# Install ansible-builder globally
pip install ansible-builder

# Or install from requirements file
pip install -r requirements-build.txt
```

Note: Global installation may cause dependency conflicts with other Python projects on your system.

Virtual Environment Management

After completing your work, you can deactivate the virtual environment:

```
# Deactivate virtual environment
deactivate
```

To reactivate the virtual environment in future sessions:

```
# Navigate to the project directory
cd /path/to/os-migrate  # or vmware-migration-kit
```

```
# Activate the virtual environment
source .venv/bin/activate
```

Troubleshooting Virtual Environment Issues

Virtual environment not found

Ensure you’re in the correct directory and the virtual environment was created successfully.

Permission denied

On some systems, you may need to use `python3` instead of `python` to create the virtual environment.

Dependencies not found

Make sure the virtual environment is activated before installing dependencies or building AEE images.

```
# Check if virtual environment is active
echo $VIRTUAL_ENV

# Verify ansible-builder is installed
which ansible-builder
ansible-builder --version
```

Building os-migrate AEE

Navigate to the os-migrate repository and build the AEE:

```
# Navigate to the repository
cd /path/to/os-migrate

# Activate virtual environment (if using one)
source .venv/bin/activate

# Navigate to AEE directory
cd aee

# Build the AEE image
ansible-builder build --tag os-migrate:latest
```

Building vmware-migration-kit AEE

Navigate to the vmware-migration-kit repository and build the AEE:

```
# Navigate to the repository
cd /path/to/vmware-migration-kit

# Activate virtual environment (if using one)
```

```
source .venv/bin/activate

# Navigate to AEE directory
cd aee

# Build the AEE image
ansible-builder build --tag vmware-migration-kit:latest
```

Automated Build Process

Both repositories include GitHub Actions workflows that automatically build and test AEE images:

- `os-migrate/.github/workflows/build-aee.yml`
- `vmware-migration-kit/.github/workflows/build-aee.yml`

These workflows:

- Trigger on pushes to main branch and pull requests
- Build the AEE image using ansible-builder
- Run basic validation tests
- Push images to container registries (when configured)

Release Versioning and Tagging Strategy

The GitHub Actions workflows implement a sophisticated versioning strategy for AEE images:

Main Branch Builds

Images built from the `main` branch are tagged as `latest`:

```
# When building from main branch
- name: Build and push AEE image
  if: github.ref == 'refs/heads/main'
  run: |
    ansible-builder build --tag ${{ github.repository }}:latest
    podman push ${{ github.repository }}:latest
```

Tag-based Builds

When building from Git tags, images receive multiple tags for maximum compatibility:

```
# When building from tags
- name: Build and push AEE image with version tags
  if: startsWith(github.ref, 'refs/tags/')
  run: |
    TAG_VERSION=${GITHUB_REF#refs/tags/}
    ansible-builder build --tag ${{ github.repository }}:$TAG_VERSION
```

```
    ansible-builder build --tag ${{ github.repository }}:stable

    podman push ${{ github.repository }}:$TAG_VERSION
    podman push ${{ github.repository }}:stable
```

Tagging Strategy

The versioning strategy follows these rules:

- `latest` - Always points to the most recent build from `main` branch
- `stable` - Points to the most recent tagged release (production-ready)
- `1.2.3` - Version without 'v' prefix for compatibility

Usage Examples

Use the appropriate tag based on your requirements:

```
# Use latest development version
podman run --rm os-migrate:latest ansible --version

# Use latest stable release
podman run --rm os-migrate:stable ansible --version

# Use specific version
podman run --rm os-migrate:1.2.3 ansible --version
```

Workflow Triggers

The GitHub Actions workflows are triggered by:

- `push` to `main` branch → builds `latest` tag
- `push` of tags → builds version-specific and `stable` tags
- `pull_request` to `main` → builds and tests (no push to registry)

Registry Configuration

Configure the container registry in the workflow using environment variables and secrets:

```
env:
  REGISTRY: quay.io
  IMAGE_NAME: os-migrate/os-migrate

- name: Login to Container Registry
  run: |
    podman login -u ${{ secrets.REGISTRY_USERNAME }} \
                 -p ${{ secrets.REGISTRY_PASSWORD }} \
                 ${{ env.REGISTRY }}
```

```
- name: Build and Push
  run: |
    ansible-builder build --tag ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{
steps.version.outputs.tag }}
    podman push ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{
steps.version.outputs.tag }}
```

Configuring Secrets and Environment Variables

GitHub Actions supports multiple levels of configuration for secrets and variables. Understanding these levels is crucial for proper AEE workflow configuration.

Repository-Level Secrets

Create secrets at the repository level for AEE workflows:

1. Navigate to your repository on GitHub
2. Click **Settings** → **Secrets and variables** → **Actions**
3. Click **New repository secret**
4. Add the following secrets for AEE workflows:

```
# Required secrets for AEE workflows
REGISTRY_USERNAME: your-registry-username
REGISTRY_PASSWORD: your-registry-password
REGISTRY_TOKEN: your-registry-token  # Alternative to username/password
```

Environment-Level Secrets

For production deployments, use environment-level secrets:

1. Go to **Settings** → **Environments**
2. Create environments like `production`, `staging`, `development`
3. Configure environment-specific secrets:

```
# Environment-specific secrets
production:
  REGISTRY_USERNAME: prod-registry-user
  REGISTRY_PASSWORD: prod-registry-password

staging:
  REGISTRY_USERNAME: staging-registry-user
  REGISTRY_PASSWORD: staging-registry-password
```

Organization-Level Variables

Use organization-level variables for shared configuration:

1. Go to organization **Settings** → **Secrets and variables** → **Actions**
2. Add organization variables:

```yaml
# Organization variables (not secrets)
DEFAULT_REGISTRY: quay.io
DEFAULT_IMAGE_PREFIX: os-migrate
ANSIBLE_BUILDER_VERSION: 3.0.0
```

Repository-Level Variables

Create repository-level variables for project-specific configuration:

1. Navigate to your repository on GitHub
2. Click **Settings** → **Secrets and variables** → **Actions**
3. Click **Variables** tab → **New repository variable**
4. Add variables for AEE workflows:

```yaml
# Repository variables for AEE workflows
IMAGE_NAME: os-migrate
BASE_IMAGE: quay.io/ansible/ansible-runner:latest
ANSIBLE_VERSION: 6.0.0
PYTHON_VERSION: 3.11
BUILD_CONTEXT: ./aee
```

Environment-Level Variables

Configure environment-specific variables:

1. Go to **Settings** → **Environments**
2. Select an environment (e.g., `production`)
3. Add environment-specific variables:

```yaml
# Environment-specific variables
production:
  IMAGE_TAG: latest
  REGISTRY_URL: quay.io
  BUILD_ARGS: --no-cache --compress

staging:
  IMAGE_TAG: staging
  REGISTRY_URL: ghcr.io
  BUILD_ARGS: --no-cache

development:
  IMAGE_TAG: dev
```

```
  REGISTRY_URL: ghcr.io
  BUILD_ARGS: --progress=plain
```

Using Variables in Workflows

Access variables using the `vars` context in your workflows:

```
name: AEE Build with Variables
on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    environment: production

    steps:
      - uses: actions/checkout@v4

      - name: Set up Podman
        uses: redhat-actions/setup-podman@v1

      - name: Build AEE Image
        run: |
          cd ${{ vars.BUILD_CONTEXT }}
          ansible-builder build \
            --tag ${{ vars.REGISTRY_URL }}/${{ vars.IMAGE_NAME }}:${{ vars.IMAGE_TAG
}} \
            ${{ vars.BUILD_ARGS }}

      - name: Push Image
        run: |
          podman push ${{ vars.REGISTRY_URL }}/${{ vars.IMAGE_NAME }}:${{
vars.IMAGE_TAG }}
```

Variable Precedence

GitHub Actions follows this precedence order for variables and secrets:

1. **Environment variables** (highest priority)
2. **Environment-level secrets/variables**
3. **Repository-level secrets/variables**
4. **Organization-level secrets/variables**
5. **System variables** (lowest priority)

```
# Example showing variable precedence
```

```
name: Variable Precedence Example
on: push

jobs:
  test:
    runs-on: ubuntu-latest
    environment: production

    steps:
      - name: Show Variable Values
        run: |
          echo "Repository variable: ${{ vars.IMAGE_NAME }}"
          echo "Environment variable: ${{ vars.IMAGE_TAG }}"
          echo "Organization variable: ${{ vars.DEFAULT_REGISTRY }}"
          echo "System variable: ${{ github.ref_name }}"
        env:
          # This overrides all other variables
          IMAGE_NAME: override-from-env
```

Workflow Configuration Examples

Basic Registry Authentication

```
name: Build AEE Image
on:
  push:
    branches: [main]
    tags: ['v*']

jobs:
  build:
    runs-on: ubuntu-latest
    environment: production  # Uses environment-level secrets

    steps:
      - uses: actions/checkout@v4

      - name: Set up Podman
        uses: redhat-actions/setup-podman@v1
        with:
          podman-version: latest

      - name: Login to Registry
        run: |
          echo ${{ secrets.REGISTRY_PASSWORD }} | podman login \
            --username ${{ secrets.REGISTRY_USERNAME }} \
            --password-stdin \
            ${{ vars.DEFAULT_REGISTRY }}

      - name: Build AEE Image
```

```
        run: |
          cd aee
          ansible-builder build --tag ${{ vars.DEFAULT_REGISTRY }}/${{
vars.DEFAULT_IMAGE_PREFIX }}:${{ github.ref_name }}

      - name: Push Image
        run: |
          podman push ${{ vars.DEFAULT_REGISTRY }}/${{ vars.DEFAULT_IMAGE_PREFIX
}}:${{ github.ref_name }}
```

Multi-Registry Support

```
name: Build and Push to Multiple Registries
on:
  push:
    tags: ['v*']

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        registry: [quay.io, ghcr.io, docker.io]

    steps:
      - uses: actions/checkout@v4

      - name: Set up Podman
        uses: redhat-actions/setup-podman@v1

      - name: Login to ${{ matrix.registry }}
        run: |
          case "${{ matrix.registry }}" in
            "quay.io")
              echo ${{ secrets.QUAY_TOKEN }} | podman login --username ${{
secrets.QUAY_USERNAME }} --password-stdin quay.io
              ;;
            "ghcr.io")
              echo ${{ secrets.GITHUB_TOKEN }} | podman login --username ${{
github.actor }} --password-stdin ghcr.io
              ;;
            "docker.io")
              echo ${{ secrets.DOCKERHUB_TOKEN }} | podman login --username ${{
secrets.DOCKERHUB_USERNAME }} --password-stdin docker.io
              ;;
          esac

      - name: Build and Push
        run: |
          cd aee
```

```
          ansible-builder build --tag ${{ matrix.registry }}/os-migrate:${{
github.ref_name }}
          podman push ${{ matrix.registry }}/os-migrate:${{ github.ref_name }}
```

Secure Secret Handling

Follow security best practices when using secrets:

```
- name: Secure Secret Usage
  run: |
    # □ Good: Use environment variables
    export REGISTRY_PASSWORD="${{ secrets.REGISTRY_PASSWORD }}"
    podman login --username ${{ secrets.REGISTRY_USERNAME }} --password-stdin ${{
env.REGISTRY }}

    # □ Good: Use proper quoting
    podman login --username "${{ secrets.REGISTRY_USERNAME }}" --password "${{
secrets.REGISTRY_PASSWORD }}" ${{ env.REGISTRY }}

    # □ Bad: Direct command line usage without quoting
    podman login --username ${{ secrets.REGISTRY_USERNAME }} --password ${{
secrets.REGISTRY_PASSWORD }} ${{ env.REGISTRY }}
  env:
    REGISTRY: ${{ vars.DEFAULT_REGISTRY }}
```

Conditional Secret Usage

Use secrets conditionally based on workflow context:

```
- name: Conditional Registry Login
  if: github.event_name == 'push' && github.ref == 'refs/heads/main'
  run: |
    echo ${{ secrets.REGISTRY_PASSWORD }} | podman login \
      --username ${{ secrets.REGISTRY_USERNAME }} \
      --password-stdin \
      ${{ env.REGISTRY }}

- name: Build and Push (Main Branch)
  if: github.event_name == 'push' && github.ref == 'refs/heads/main'
  run: |
    cd aee
    ansible-builder build --tag ${{ env.REGISTRY }}/os-migrate:latest
    podman push ${{ env.REGISTRY }}/os-migrate:latest

- name: Build and Push (Tags)
  if: github.event_name == 'push' && startsWith(github.ref, 'refs/tags/')
  run: |
    cd aee
    TAG_VERSION=${GITHUB_REF#refs/tags/}
```

```
    ansible-builder build --tag ${{ env.REGISTRY }}/os-migrate:$TAG_VERSION
    ansible-builder build --tag ${{ env.REGISTRY }}/os-migrate:stable
    podman push ${{ env.REGISTRY }}/os-migrate:$TAG_VERSION
    podman push ${{ env.REGISTRY }}/os-migrate:stable
```

Secret Rotation and Management

Implement secret rotation strategies:

```yaml
- name: Validate Secrets
  run: |
    if [ -z "${{ secrets.REGISTRY_USERNAME }}" ]; then
      echo "□ REGISTRY_USERNAME secret is not set"
      exit 1
    fi

    if [ -z "${{ secrets.REGISTRY_PASSWORD }}" ]; then
      echo "□ REGISTRY_PASSWORD secret is not set"
      exit 1
    fi

    echo "□ All required secrets are configured"

- name: Test Registry Access
  run: |
    echo ${{ secrets.REGISTRY_PASSWORD }} | podman login \
      --username ${{ secrets.REGISTRY_USERNAME }} \
      --password-stdin \
      ${{ env.REGISTRY }} --test
    echo "□ Registry authentication successful"
```

Environment-Specific Configuration

Use different configurations for different environments:

```yaml
name: AEE Build Matrix
on:
  push:
    branches: [main, develop]
    tags: ['v*']

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        include:
          - environment: development
            registry: ghcr.io
```

```
            image_tag: dev
          - environment: staging
            registry: quay.io
            image_tag: staging
          - environment: production
            registry: quay.io
            image_tag: latest

    environment: ${{ matrix.environment }}

    steps:
      - uses: actions/checkout@v4

      - name: Set up Podman
        uses: redhat-actions/setup-podman@v1

      - name: Login to Registry
        run: |
          echo ${{ secrets.REGISTRY_PASSWORD }} | podman login \
            --username ${{ secrets.REGISTRY_USERNAME }} \
            --password-stdin \
            ${{ matrix.registry }}

      - name: Build AEE Image
        run: |
          cd aee
          ansible-builder build --tag ${{ matrix.registry }}/os-migrate:${{
matrix.image_tag }}

      - name: Push Image
        run: |
          podman push ${{ matrix.registry }}/os-migrate:${{ matrix.image_tag }}
```

4.10.4. Using AEE Images

Running Playbooks with AEE

Execute os-migrate playbooks using the AEE container:

```
podman run --rm -it \
  -v $(pwd):/runner \
  -v ~/.ssh:/home/runner/.ssh:ro \
  os-migrate:latest \
  ansible-playbook -i inventory playbook.yml
```

Interactive Shell Access

Access the AEE container interactively for debugging:

```
podman run --rm -it \
  -v $(pwd):/runner \
  -v ~/.ssh:/home/runner/.ssh:ro \
  os-migrate:latest \
  /bin/bash
```

Volume Mounts

Common volume mounts for AEE usage:

- `$(pwd):/runner` - Mount current directory as working directory
- `~/.ssh:/home/runner/.ssh:ro` - Mount SSH keys (read-only)
- `~/.config/openstack:/home/runner/.config/openstack:ro` - Mount OpenStack credentials
- `/path/to/inventory:/runner/inventory:ro` - Mount inventory files

4.10.5. AEE Configuration

Execution Environment Definition

AEE images are defined using `execution-environment.yml` files that specify:

- Base image (typically `quay.io/ansible/ansible-runner:latest`)
- Python dependencies
- Ansible collections
- Additional system packages

Example structure:

```
version: 1
dependencies:
  galaxy:
    - name: os_migrate.os_migrate
      source: https://github.com/os-migrate/os-migrate
  python:
    - openstacksdk>=1.0.0
    - ansible>=6.0.0
  system:
    - git
    - openssh-clients
```

Customizing AEE Images

To customize AEE images for specific requirements:

1. Modify the `execution-environment.yml` file

2. Add custom requirements or collections
3. Rebuild the image using ansible-builder

```
ansible-builder build --tag custom-aee:latest
```

4.10.6. Troubleshooting

Secrets and Variables Issues

Common Secret Configuration Problems

Secret Not Found

Ensure the secret is created at the correct level (repository, environment, or organization) and the name matches exactly in the workflow.

Permission Denied

Verify that the workflow has access to the environment containing the secrets. Check environment protection rules and required reviewers.

Empty Secret Value

Secrets that are not set return empty strings. Always validate secret existence before use.

```
- name: Validate Required Secrets
  run: |
    if [ -z "${{ secrets.REGISTRY_USERNAME }}" ]; then
      echo "▯ REGISTRY_USERNAME secret is not configured"
      exit 1
    fi

    if [ -z "${{ secrets.REGISTRY_PASSWORD }}" ]; then
      echo "▯ REGISTRY_PASSWORD secret is not configured"
      exit 1
    fi

    echo "▯ All required secrets are available"
```

Variable Access Issues

Variable Not Defined

Check that variables are created at the appropriate level and use the correct context (`vars` for variables, `secrets` for secrets).

Wrong Variable Context

Use `${{ vars.VARIABLE_NAME }}` for variables and `${{ secrets.SECRET_NAME }}` for secrets.

```
- name: Debug Variable Access
```

```
  run: |
    echo "Repository variables:"
    echo "  IMAGE_NAME: ${{ vars.IMAGE_NAME }}"
    echo "  BUILD_CONTEXT: ${{ vars.BUILD_CONTEXT }}"

    echo "Environment variables:"
    echo "  IMAGE_TAG: ${{ vars.IMAGE_TAG }}"
    echo "  REGISTRY_URL: ${{ vars.REGISTRY_URL }}"

    echo "Organization variables:"
    echo "  DEFAULT_REGISTRY: ${{ vars.DEFAULT_REGISTRY }}"
```

Registry Authentication Troubleshooting

Authentication Failed

Verify credentials are correct and have appropriate permissions for the registry.

Token Expired

Check if the registry token has expired and needs renewal.

```
- name: Test Registry Authentication
  run: |
    echo "Testing authentication to ${{ vars.DEFAULT_REGISTRY }}"

    # Test login without pushing
    echo ${{ secrets.REGISTRY_PASSWORD }} | podman login \
      --username ${{ secrets.REGISTRY_USERNAME }} \
      --password-stdin \
      ${{ vars.DEFAULT_REGISTRY }} --test

    if [ $? -eq 0 ]; then
      echo "▯ Registry authentication successful"
    else
      echo "▯ Registry authentication failed"
      exit 1
    fi
```

Debugging AEE Issues

Enable verbose output for troubleshooting:

```
podman run --rm -it \
  -v $(pwd):/runner \
  os-migrate:latest \
  ansible-playbook -vvv -i inventory playbook.yml
```

Check container logs:

```
podman logs <container_id>
```

Performance Optimization

- Use volume mounts instead of copying files into containers
- Mount only necessary directories to reduce I/O overhead
- Consider using read-only mounts where possible
- Use appropriate resource limits for container execution

4.10.7. Maintenance

Updating AEE Images

Regular updates ensure security and compatibility:

1. Update base images in execution environment definitions
2. Update Ansible collections to latest versions
3. Update Python dependencies
4. Rebuild and test AEE images
5. Update documentation with any changes

Version Management

The automated GitHub Actions workflows handle version management based on Git references:

Manual Version Management

For local development, you can manually tag images:

```
# Build specific version locally
ansible-builder build --tag os-migrate:1.2.3

# Build latest development version
ansible-builder build --tag os-migrate:latest
```

Automated Version Management

The GitHub Actions workflows automatically handle versioning:

- **Main branch pushes** → `latest` tag
- **Tag pushes** → version-specific tag + `stable` tag
- **Pull requests** → build and test only (no registry push)

Creating Releases

To create a new release:

1. Create and push a Git tag: [source,bash] ---- git tag -a 1.2.3 -m "Release version 1.2.3" git push origin 1.2.3 ----
2. The GitHub Actions workflow will automatically:
    - Build the AEE image
    - Tag it with `1.2.3` and `stable`
    - Push to the configured registry

Version Tag Strategy

- `latest` - Development builds from main branch
- `stable` - Latest tagged release (production-ready)
- `1.2.3` - Specific version

Security Considerations

- Regularly update base images to include security patches
- Scan AEE images for vulnerabilities
- Use minimal base images when possible
- Review and audit all included dependencies

4.10.8. Best Practices

Development Workflow

1. Test changes locally using AEE containers
2. Use version-controlled execution environment definitions
3. Document any customizations or modifications
4. Test AEE images in target environments before deployment

Production Usage

1. Use specific version tags instead of `latest`
2. Implement proper monitoring and logging
3. Regular security updates and vulnerability scanning
4. Backup and disaster recovery planning

Documentation

1. Keep execution environment definitions well-documented
2. Document any custom modifications or extensions

3. Provide clear usage examples and troubleshooting guides
4. Maintain compatibility matrices for different versions

4.10.9. TODO

Collection Installation Improvements

Improve the way collections (os-migrate or vmware-migration-kit) are installed within AEE images to ensure proper version alignment:

- **Main branch builds**: When the image tag is `main`, install the collection content directly from the main branch repository as the source of installation using Git-based requirements
- **Stable/tagged builds**: When the image tag is `stable` or matches a repository tag, ensure the installation uses the corresponding tagged version of the collection from Ansible Galaxy
- **Dynamic requirements.yml**: Implement automated generation of `requirements.yml` files based on build context to ensure proper collection versioning
- **Version consistency validation**: Add build-time checks to verify that AEE image tags match the collection versions they contain

This improvement will ensure that AEE images always contain the correct version of the collection that matches the build context and tag strategy, providing better reproducibility and version alignment.

4.11. Migration Guides

4.12. OS Migrate walkthrough

OS Migrate is a framework for OpenStack parallel cloud migration (migrating content between OpenStack tenants which are not necessarily in the same cloud). It's a collection of Ansible playbooks that provide the basic functionality, but may not fit each use case out of the box. You can craft custom playbooks using the OS Migrate collection pieces (roles and modules) as building blocks.

Parallel cloud migration is a way to modernize an OpenStack deployment. Instead of upgrading an OpenStack cluster in place, a second OpenStack cluster is deployed alongside, and tenant content is migrated from the original cluster to the new one. Parallel cloud migration is best suited for environments which are due for a hardware refresh cycle. It may also be performed without a hardware refresh, but extra hardware resources are needed to bootstrap the new cluster. As hardware resources free up in the original cluster, they can be gradually added to the new cluster.

OS Migrate strictly uses the official OpenStack API and does not utilize direct database access or other methods to export or import data. The Ansible playbooks contained in OS Migrate are idempotent. If a command fails, you can retry with the same command.

The migration is generally performed in this sequence:

- prerequisites: prepare authentication info, parameter files,

- pre-workload migration, which copies applicable resources into the destination cloud (e.g. networks, security groups, images) while workloads keep running in the source cloud,
- workload migration, which stops usage of applicable resources in the source cloud and moves them into the destination cloud (VMs, volumes).

4.12.1. Prerequisites

Authentication

Users are encouraged to use os-migrate using specific credentials for each project/tenant, this means **not using the admin user to execute the resources migration** (unless the resource is owned by the admin project, e.g. public Glance images).

In case the circumstances require migrating by the `admin` user, this user needs to have access to the respective projects. There are two options:

- Add the `admin` user as a `member` of each project.

  ```
  Depending on how many projects need to be migrated this approach seems
  to be suboptimal as there are involved several configuration updates in
  the projects that will need to be reverted after the migration
  completes.
  ```

- Create a group including the admin user and add the group to each project as member.

  ```
  The difference with this approach is that once the migration is
  completed, by removing the group, all the references in all the projects
  will be removed automatically.
  ```

Parameter file

Let’s create an `os-migrate-vars.yml` file with Ansible variables:

YAML Ansible Variables

```yaml
os_migrate_src_auth:
  auth_url: http://192.168.0.13.199/v3
  password: srcpassword
  project_domain_name: Default
  project_name: src
  user_domain_name: Default
  username: src
os_migrate_src_region_name: regionOne
os_migrate_dst_auth:
  auth_url: http://192.167.0.16:5000/v3
  password: dstpassword
  project_domain_name: Default
  project_name: dst
```

```
      user_domain_name: Default
      username: dst
    os_migrate_dst_region_name: regionOne
    os_migrate_data_dir: /home/migrator/os-migrate-data
```

The file contains the source and destination tenant credentials, a directory on the migrator host (typically localhost) and a directory where the exported data will be saved.

If you are migrating content from multiple source projects, make sure to use a separate data directory for each source project. In other words, when changing `os_migrate_src_auth` or `os_migrate_src_region_name`, make sure to also change `os_migrate_data_dir`.

A note about Keystone v2

As depicted in content of the previously defined `os-migrate-vars.yml` file, the parameters `os_migrate_src_auth` and `os_migrate_dst_auth` refer to the usage of Keystone v3. In the case of a user needing to execute a migration between tenants not supporting Keystone v3 the following error will be raised:

```
keystoneauth1.exceptions.discovery.DiscoveryFailure: Cannot use v2 authentication with
domain scope
```

To fix this issue, the user must adjust their auth parameters:

YAML Keystone V2 Auth Params

```
    os_migrate_src_auth:
      auth_url: http://192.168.0.13.199/v2.0
      password: srcpassword
      project_name: src
      username: src
    os_migrate_src_region_name: regionOne
```

Notice that the parameters `project_domain_name` and `user_domain_name` are removed and the `auth_url` parameter points to the Keystone v2 endpoint.

Shortcuts

We will use the OS Migrate collection path and an ansible-playbook command with the following arguments routinely, so let's save them as variables in the shell:

```
export
OSM_DIR=/home/migrator/.ansible/collections/ansible_collections/os_migrate/os_migrate
export OSM_CMD="ansible-playbook -v -i $OSM_DIR/localhost_inventory.yml -e @os-
migrate-vars.yml"
```

Pre-workload migration

Workloads require the support of several resources in a given cloud to operate properly. Some of these resources include networks, subnets, routers, router interfaces, security groups, and security group rules. The pre-workload migration process includes exporting these resources from the source cloud onto the migrator machine, the option to edit the resources if desired, and importing them into the destination cloud.

Exporting or importing resources is enabled by running the corresponding playbook from OS Migrate. Let’s look at a concrete example. To export the networks, run the "export_networks" playbook.

Export and import

To export the networks:

```
$OSM_CMD $OSM_DIR/playbooks/export_networks.yml
```

This will create networks.yml file in the data directory, similar to this:

Networks YAML example

```
os_migrate_version: 0.17.0
resources:
  - _info:
      availability_zones:
        - nova
      created_at: '2020-04-07T14:08:30Z'
      id: a1eb31f6-2cdc-4896-b582-8950dafa34aa
      project_id: 2f444c71265048f7a9d21f81db6f21a4
      qos_policy_id: null
      revision_number: 3
      status: ACTIVE
      subnet_ids:
        - a5052e10-5e00-432b-a826-29695677aca0
        - d450ffd0-972e-4398-ab49-6ba9e29e2499
      updated_at: '2020-04-07T14:08:34Z'
    params:
      availability_zone_hints: []
      description: ''
      dns_domain: null
      is_admin_state_up: true
      is_default: null
      is_port_security_enabled: true
      is_router_external: false
      is_shared: false
      is_vlan_transparent: null
      mtu: 1450
      name: osm_net
      provider_network_type: null
```

```
          provider_physical_network: null
          provider_segmentation_id: null
          qos_policy_name: null
          segments: null
        type: openstack.network.Network
```

You may edit the file as needed and then run the "import_networks" playbook to import the networks from this file into the destination cloud:

```
$OSM_CMD $OSM_DIR/playbooks/import_networks.yml
```

You can repeat this process for other resources like subnets, security groups, security group rules, routers, router interfaces, images and keypairs.

For a full list of available playbooks, run:

```
ls $OSM_DIR/playbooks
```

Diagrams

///TODO need to figure out these UMLs in the source .. figure:: ../images/render/pre-workload-migration-workflow.png :alt: Pre-workload Migration (workflow) :width: 50%

```
Pre-workload Migration (workflow)
```

a. figure:: ../images/render/pre-workload-migration-data-flow.png :alt: Pre-workload Migration (data flow) :width: 50%

   ```
   Pre-workload Migration (data flow)
   ```

Demo

///TODO: Video link `Pre-workload migration recorded demo https://youtu.be/e7KXy5Hq4CMA`_:

|Watch the video1|

Workload migration

Workload information is exported in a similar method to networks, security groups, etc. as in the previous sections. Run the "export_workloads" playbook, and edit the resulting workloads.yml as desired:

```
os_migrate_version: 0.17.0
resources:
- _info:
```

```
       addresses:
         external_network:
         - OS-EXT-IPS-MAC:mac_addr: fa:16:3e:98:19:a0
           OS-EXT-IPS:type: fixed
           addr: 10.19.2.41
           version: 4
       flavor_id: a96b2815-3525-4eea-9ab4-14ba58e17835
       id: 0025f062-f684-4e02-9da2-3219e011ec74
       status: SHUTOFF
     params:
       flavor_name: m1.small
       name: migration-vm
       security_group_names:
       - testing123
       - default
     type: openstack.compute.Server
```

Note that this playbook only extracts metadata about servers in the specified tenant - it does not download OpenStack volumes directly to the migration data directory. Data transfer is handled by the import_workloads playbook. The data is transfered directly between the clouds, meaning both clouds have to be running and reachable at the same time. The following sections describe the process in more detail.

Process Summary

This flowchart illustrates the high-level migration workflow, from a user's point of view:

///TODO UML image .. figure:: ../images/render/workload-migration-workflow.png :alt: Workload migration (workflow) :width: 50%

```
Workload migration (workflow)
```

The process involves the deployment of a "conversion host" on source and destination clouds. A conversion host is an OpenStack server which will be used to transfer binary volume data from the source to the destination cloud. The conversion hosts are expected to be created from CentOS 9 or RHEL 8 cloud images.

The following diagram helps explain the need for a conversion host VM:

///TODO UML image .. figure:: ../images/render/workload-migration-data-flow.png :alt: Workload migration (data flow) :width: 80%

```
Workload migration (data flow)
```

This shows that volumes on the source and destination clouds are removed from their original VMs and attached to their respective conversion hosts, and then transferred over the network from the source conversion host to the destination. The tooling inside the conversion host migrates one server by automating these actions on the source and destination clouds:

Source Cloud:

- Detach volumes from the target server to migrate
- Attach the volumes to the source conversion host
- Export the volumes as block devices and wait for destination conversion host to connect

Destination Cloud:

- Create new volumes on the destination conversion host, one for each source volume
- Attach the new volumes to the destination conversion host
- Connect to the block devices exported by source conversion host, and copy the data to the new attached volumes
- Detach the volumes from the destination conversion host
- Create a new server using the new volumes

This method keeps broad compatibility with the various flavors and configurations of OpenStack using as much of an API-only approach as possible, while allowing the use of libguestfs-based tooling to minimize total data transfer.

Preparation

We'll put additional parameters into `os-migrate-vars.yml`:

```
os_migrate_conversion_external_network_name: public
os_migrate_conversion_flavor_name: m1.large
```

These define the flavor and external network we want to use for our conversion hosts.

By default the migration will use an image named `os_migrate_conv` for conversion hosts. Make sure this image exists in Glance on both clouds. Currently it should be a CentOS 9 Cloud Image or RHEL 8 KVM Guest Image

When using RHEL as conversion host, make sure to set the necessary RHEL Variables

Conversion host deployment

The conversion host deployment playbook creates the servers, installs additional required packages, and authorizes the destination conversion host to connect to the source conversion host for the actual data transfer.

```
$OSM_CMD $OSM_DIR/playbooks/deploy_conversion_hosts.yml
```

Export

Before migrating workloads, the destination cloud must have imported all other resources (networks, security groups, etc.) or the migration will fail. Matching named resources (including flavor names) must exist on the destination before the servers are created.

Export workload information with the export_workloads playbook. Each server listed in the resulting workloads.yml will be migrated, except for the one matching the name given to the source conversion host server.

```
$OSM_CMD $OSM_DIR/playbooks/export_workloads.yml
```

The resulting workloads.yml file will look similar to:

```
os_migrate_version: 0.17.0
resources:
- _info:
    created_at: '2020-11-12T17:55:40Z'
    flavor_id: cd6258f9-c34b-4a9c-a1e2-8cb81826781e
    id: af615f8c-378a-4a2e-be6a-b4d38a954242
    launched_at: '2020-11-12T17:56:00.000000'
    security_group_ids:
    - 1359ec88-4873-40d2-aa0b-18ad0588f107
    status: SHUTOFF
    updated_at: '2020-11-12T17:56:30Z'
    user_id: 48be0a2e86a84682b8e4992a65d39e3e
  _migration_params:
    boot_disk_copy: false
  params:
    availability_zone: nova
    config_drive: null
    description: osm_server
    disk_config: MANUAL
    flavor_ref:
      domain_name: null
      name: m1.xtiny
      project_name: null
    image_ref:
      domain_name: null
      name: cirros-0.4.0-x86_64-disk.img
      project_name: null
    key_name: osm_key
    metadata: {}
    name: osm_server
    ports:
    - _info:
        device_id: af615f8c-378a-4a2e-be6a-b4d38a954242
        device_owner: compute:nova
        id: cf5d73c3-089b-456b-abb9-dc5da988844e
      _migration_params: {}
      params:
        fixed_ips_refs:
        - ip_address: 192.168.20.7
```

```
             subnet_ref:
               domain_name: '%auth%'
               name: osm_subnet
               project_name: '%auth%'
           network_ref:
             domain_name: '%auth%'
             name: osm_net
             project_name: '%auth%'
         type: openstack.network.ServerPort
       scheduler_hints: null
       security_group_refs:
       - domain_name: '%auth%'
         name: osm_security_group
         project_name: '%auth%'
       tags: []
       user_data: null
     type: openstack.compute.Server
```

Migration parameters

///TODO next chapter, inc. link You can edit the exported `workloads.yml` to adjust desired properties for the servers which will be created in the destination cloud during migration. You can also edit migration parameters to control how a workload should be migrated. Refer to `Migration Parameters Guide <migration-params-guide.html>`_ for more information.

Ansible Variables

///TODO next chapter, inc. link In addition to the migration parameters in the resource YAML files, you can alter the behavior of OS Migrate via Ansible variables, e.g. to specify a subset of resources/workloads that will be exported or imported. Refer to the `Variables Guide <variables-guide.html>`_ for details.

Migration

Then run the import_workloads playbook to migrate the workloads:

```
$OSM_CMD $OSM_DIR/playbooks/import_workloads.yml
```

Any server marked "changed" should be successfully migrated to the destination cloud. Servers are "skipped" if they match the name or ID of the specified conversion host. If there is already an server on the destination matching the name of the current server, it will be marked "ok" and no extra work will be performed.

Cleanup of conversion hosts

When you are done migrating workloads in given tenants, delete their conversion hosts via the delete_conversion_hosts playbook:

```
$OSM_CMD $OSM_DIR/playbooks/delete_conversion_hosts.yml
```

Demo

Workload migration recorded demo

4.13. How it works: Workload migration

The information described in this page is not necessary knowledge for using OS Migrate workload migration. However, knowing how the migration works under the hood may be useful for troubleshooting or forming deeper understanding of OS Migrate.

4.13.1. Data flow

The workload migration utilizes so called **conversion hosts** to transfer data from the source cloud to the destination cloud.

The conversion hosts are temporarily deployed in the source & destination projects. Being in the same project as the source (and destination, respectively) VMs ensures that the conversion hosts will have access to the data that needs to be migrated (snapshots and volumes).

a. figure:: ../images/render/workload-migration-data-flow.png :alt: Workload migration (data flow) :width: 80%

   ```
   Workload migration (data flow)
   ```

4.13.2. Migration sequence

The `export_workloads.yml` playbook simply exports workload metadata into `workloads.yml`.

The actual main migration sequence happens inside `import_workloads.yml` playbook and the `import_workloads` role. The initial common steps are:

- The resources loaded from `workloads.yml` are validated.
- Resources are filtered according to `os_migrate_workloads_filter`.
- Reachability of source & destination conversion hosts is verified.

If you’re defaulting to the default storage migration mode `data_copy` then the role starts iterating over all workloads that passed the filter. The steps performed for each workload (Nova Server) are:

- The `import_workload_prelim` module creates log and state files under `{{ os_migrate_data_dir }}/workload_logs`. It also takes care of skipping migration of VMs that already exist in the destination, and skipping of conversion hosts, should such migration be attempted.
- The `import_workload_dst_check` module checks whether migration prerequisites are satisfied in the destination cloud/project. This means verifying that resources which are referenced by name from the workload serialization can be de-referenced in the destination cloud. In other

words, this verifies the networks, subnets etc., that the destination VM should be attached to, indeed exist in the destination cloud.

- If `os_migrate_workload_stop_before_migration` is `true`, the VM in the source cloud is stopped.
- The `import_workload_src_check` checks whether the source workload is ready to be migrated. This means verifying that the Nova Server is `SHUTOFF`.
- The `import_workload_export_volumes` module prepares data for transfer to the destination cloud:
    - If `boot_disk_copy` is `true`, a snapshot of the source VM is created, converted to a Cinder volume and attached to the source conversion host.
    - Additional Cinder volumes attached to the source VM are detached from it and attached to the source conversion host.
    - All VM's volumes (boot & additional) on the conversion host are exported as NBD drives, listening on localhost only.
    - The `import_workload_transfer_volumes` copies data from source to destination:
    - SSH port forwarding is created for the NBD drives of the source conversion host, so that they are accessible on the destination conversion host, again on localhost only. (The data transfer mechanism could be described as "NBD over SSH".)
    - Cinder volumes are created in the destination project for both the boot disk and additional volumes (as applicable). The destination volume sizes match the volume sizes in the source cloud. The volumes are attached to the destination conversion host.
    - Sparsification of the NBDs is performed, only for recognizable filesystems that the `virt-sparsify` tool supports. This significantly speeds up copying of empty space on supported filesystems.
    - Data is copied from the NBDs to the respective destination Cinder volumes.
    - SSH port forwarding for the NBDs are closed, and volumes are detached from the destination conversion host.
- The `import_workload_create_instance` creates new Nova server in the destination cloud according to the data from the resource serialization, and using the copied Cinder volumes as applicable.
- The `import_workload_src_cleanup` cleans up after the migration in the source cloud. It closes the NBD exports, detaches volumes from the conversion host, deletes the temporary boot disk snapshot volume and re-attaches any additional volumes back onto the source VM (as applicable).
- In case of failure during the migration, the `import_workload_src_cleanup` module is executed too, and an extra `import_workload_dst_failure_cleanup` module is executed, which aims to clean up failed partial migration from the destination cloud. (In case of successful migration, no further clean up is necessary in the destination cloud.)

    a. figure:: ../images/render/workload-migration-sequence.png :alt: Workload migration (sequence) :width: 80%

    ```
    Workload migration (sequence)
    ```

4.14. OS Migrate VMware Guide

An important auxilliary function included in OS Migrate is our VMware tooling, which allows you to migrade a virtual machine from an ESXi/Vcenter environment to OpenStack environments.

The code used os-migrate Ansible collection in order to deploy conversion host and setup correctly the prerequists in the Openstack destination cloud. It also used the vmware community collection in order to gather informations from the source VMWare environment.

The Ansible collection provides different steps to scale your migration from VMWare to Openstack:

- A discovery phase where it analizes the VMWare source environment and provides collected data to help for the migration.
- A pre-migration phase where it make sure the destionation cloud is ready to perform the migration, by creating the conversion host for example or the required network if needed.
- A migration phase with different workflow where the user can basicaly scale the migration with a very number of virtual machines as entry point, or can migrate sensitive virtual machine by using a near zero down time with the change block tracking VMWare option (CBT) and so perform the virtual machine migration in two steps. The migration can also be done without conversion host.

4.14.1. Workflow

There is different ways to run the migration from VMWare to OpenStack.

- The default is by using nbdkit server with a conversion host (an Openstack instance hosted in the destination cloud). This way allow the user to use the CBT option and approach a zero downtime. It can also run the migration in one time cycle.
- The second one by using virt-v2v binding with a conversion host. Here you can use a conversion host (Openstack instance) already deployed or you can let OS-Migrate deployed a conversion host for you.
- A third way is available where you can skip the conversion host and perform the migration on a Linux machine, the volume migrated and converted will be upload a Glance image or can be use later as a Cinder volume. This way is not recommended if you have big disk or a huge amount of VMs to migrate: the performance are really slower than with the other ways.

All of these are configurable with Ansible boolean variables.

4.14.2. Features and supported OS

Features

The following features are availables:

- Discovery mode
- Network mapping
- Port creation and mac addresses mapping

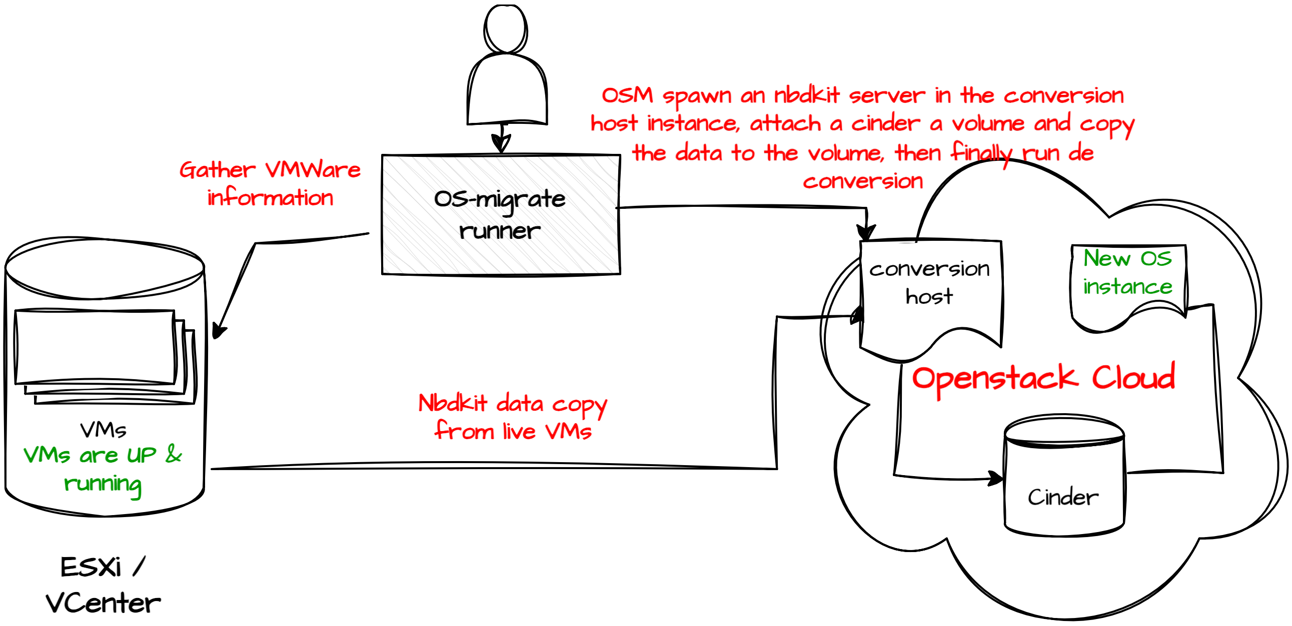
- Openstack flavor mapping and creation
- Migration with nbdkit server with change block tracking feature (CBT)
- Migration with virt-v2v
- Upload migrate volume via Glance
- Multi disks migration
- Multi nics
- Parallel migration on a same conversion host
- Ansible Automation Platform (AAP)

Supported OS

Currently we are supporting the following matrice:

OS Family	Version	Supported & Tested	Not Tested Yet
	RHEL	9.4	Yes
-	RHEL	9.3 and lower	Yes
-	RHEL	8.5	Yes
-	RHEL	8.4 and lower	-
Yes	CentOS	9	Yes
-	CentOS	8	Yes
-	Ubuntu Server	24	Yes
-	Windows	10	Yes
-	Windows Server	2k22	Yes
-	Suse	X	-

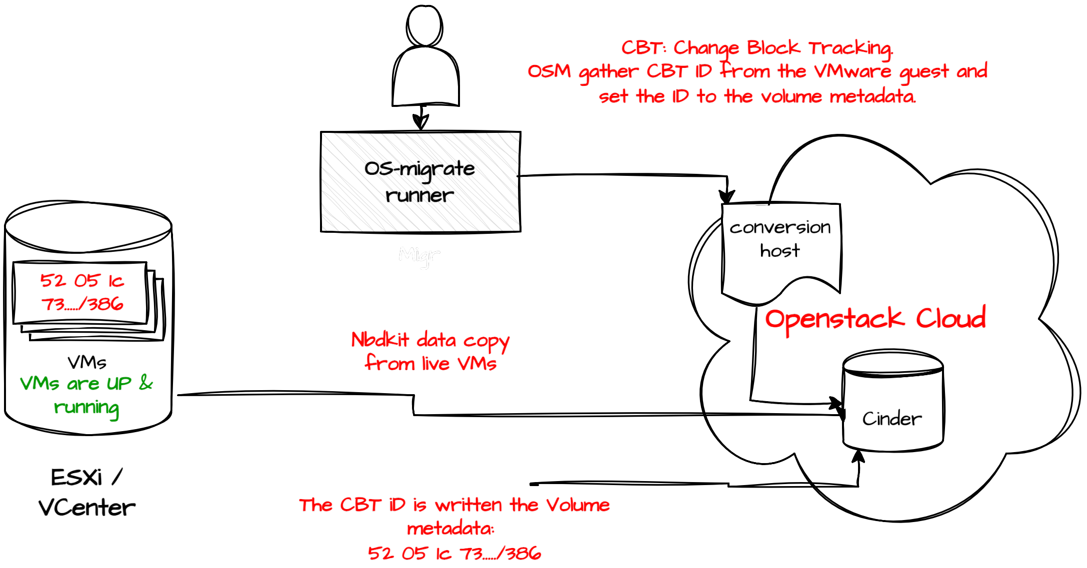
Nbdkit migration example



Nbdkit migration example with the Change Block Tracking

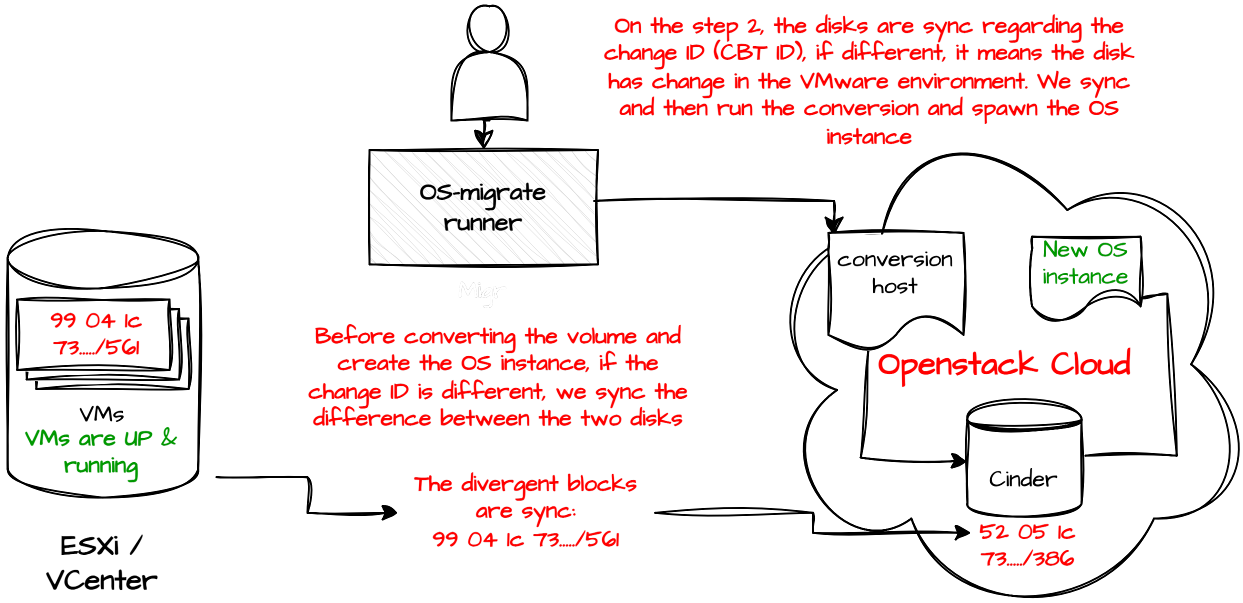
Step 1: The data are copied and the change ID from the VMware disk are set to the Cinder volume as metadata

The conversion cannot be made at this moment, and the OS instance is not created. This functionality can be used for large disks with a lot of data to transfer. It helps avoid a prolonged service interruption.



==== Step 2: OSM compare the source (VMware disk) and the destination (Openstack Volume) change ID

If the change IDs are not equal, the changed blocks between the source and destination are synced. Then, the conversion to libvirt/KVM is triggered, and the OpenStack instance is created. This allows for minimal downtime for the VMs.



=== Migration demo from an AEE

The content of the Ansible Execution Environment could be find here:

https://github.com/os-migrate/aap/blob/main/aae-container-file

And the live demo here:

Migration from VMware to OpenStack

=== Running migration

==== Conversion host

You can use os_migrate.os_migration collection to deploy a conversion, but you can easily create your conversion host manually.

A conversion host is basically an OpenStack instance.

Important: If you want to take benefit of the current supported OS, it's highly recommended to use a **CentOS-10** release or **RHEL-9.5** and superior. If you want to use other Linux distribution, make sure the virtio-win package is equal or higher than 1.40 version.

```
curl -O -k https://cloud.centos.org/centos/10-stream/x86_64/images/CentOS-Stream-GenericCloud-10-20250217.0.x86_64.qcow2

# Create OpenStack image:
openstack image create --disk-format qcow2 --file CentOS-Stream-GenericCloud-10-20250217.0.x86_64.qcow2 CentOS-Stream-GenericCloud-10-20250217.0.x86_64.qcow2

# Create flavor, security group and network if needed
openstack server create --flavor x.medium --image 14b1a895-5003-4396-888e-1fa55cd4adf8
```

```
\
  --key-name default --network private   vmware-conv-host
openstack server add floating ip vmware-conv-host 192.168.18.205
```

Inventory, Variables files and Ansible command:

inventory.yml

```
migrator:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python_interpreter: "{{ ansible_playbook_python }}"
conversion_host:
  hosts:
    192.168.18.205:
      ansible_ssh_user: cloud-user
      ansible_ssh_private_key_file: key
```

myvars.yml:

```
# if you run the migration from an Ansible Execution Environment (AEE)
# set this to true:
runner_from_aee: true

# osm working directory:
os_migrate_vmw_data_dir: /opt/os-migrate
copy_openstack_credentials_to_conv_host: false

# Re-use an already deployed conversion host:
already_deploy_conversion_host: true

# If no mapped network then set the openstack network:
openstack_private_network: 81cc01d2-5e47-4fad-b387-32686ec71fa4

# Security groups for the instance:
security_groups: ab7e2b1a-b9d3-4d31-9d2a-bab63f823243
use_existing_flavor: true
# key pair name, could be left blank
ssh_key_name: default
# network settings for openstack:
os_migrate_create_network_port: true
copy_metadata_to_conv_host: true
used_mapped_networks: false

vms_list:
  - rhel-9.4-1
```

secrets.yml:

```
# VMware parameters:
esxi_hostname: 10.0.0.7
vcenter_hostname: 10.0.0.7
vcenter_username: root
vcenter_password: root
vcenter_datacenter: Datacenter

os_cloud_environ: psi-rhos-upgrades-ci
dst_cloud:
  auth:
    auth_url: https://keystone-public-openstack.apps.ocp-4-16.standalone
    username: admin
    project_id: xyz
    project_name: admin
    user_domain_name: Default
    password: openstack
  region_name: regionOne
  interface: public
  insecure: true
  identity_api_version: 3
```

Ansible command:

```
ansible-playbook -i inventory.yml os_migrate.vmware_migration_kit.migration -e
@secrets.yml -e @myvars.yml
```

4.14.3. Usage

You can find a "how to" here, to start from sratch with a container: https://gist.github.com/matbu/003c300fd99ebfbf383729c249e9956f

Clone repository or install from ansible galaxy

```
git clone https://github.com/os-migrate/vmware-migration-kit
ansible-galaxy collection install os_migrate.vmware_migration_kit
```

Nbdkit (default)

Edit vars.yaml file and add our own setting:

```
esxi_hostname: ********
vcenter_hostname: *******
vcenter_username: root
vcenter_password: *****
```

```
vcenter_datacenter: Datacenter
```

If you already have a conversion host, or if you want to re-used a previously deployed one:

```
already_deploy_conversion_host: true
```

Then specify the Openstack credentials:

```
# OpenStack destination cloud auth parameters:
dst_cloud:
  auth:
    auth_url: https://openstack.dst.cloud:13000/v3
    username: tenant
    project_id: xyz
    project_name: migration
    user_domain_name: osm.com
    password: password
  region_name: regionOne
  interface: public
  identity_api_version: 3

# OpenStack migration parameters:
# Use mapped networks or not:
used_mapped_networks: true
network_map:
  VM Network: private

# If no mapped network then set the openstack network:
openstack_private_network: 81cc01d2-5e47-4fad-b387-32686ec71fa4

# Security groups for the instance:
security_groups: 4f077e64-bdf6-4d2a-9f2c-c5588f4948ce
use_existing_flavor: true

os_migrate_create_network_port: false

# OS-migrate parameters:
# osm working directory:
os_migrate_vmw_data_dir: /opt/os-migrate

# Set this to true if the Openstack "dst_cloud" is a clouds.yaml file
# other, if the dest_cloud is a dict of authentication parameters, set
# this to false:
copy_openstack_credentials_to_conv_host: false

# Teardown
# Set to true if you want osm to delete everything on the destination cloud.
os_migrate_tear_down: true
```

```
# VMs list
vms_list:
  - rhel-1
  - rhel-2
```

Running migration from local shared NFS

OS-Migrate can migrate directly from a local shared directory mounted on the conversion host. If the VMware virtual machines are located on an NFS datastore that is accessible to the conversion host, you can mount the NFS storage on the conversion host and provide the path to the NFS mount point.

OS-Migrate will then directly consume the disks of the virtual machines located on the NFS mount point. Configure the Ansible variable to specify your mount point as follows:

```
import_workloads_local_disk_path: "/srv/nfs"
```

In this mode, only cold migration is supported.

Ansible configuration

Create an invenvoty file, and replace the conv_host_ip by the ip address of your conversion host:

```
migrator:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python_interpreter: "{{ ansible_playbook_python }}"
conversion_host:
  hosts:
    conv_host_ip:
      ansible_ssh_user: cloud-user
      ansible_ssh_private_key_file: /home/stack/.ssh/conv-host
```

Then run the migration with:

```
ansible-playbook -i localhost_inventory.yml os_migrate.vmware_migration_kit.migration
-e @vars.yaml
```

Running Migration outside of Ansible

You can also run migration outside of Ansible because the Ansible module are written in Golang. The binaries are located in the plugins directory.

From your conversion host (or an Openstack instance inside the destination cloud) you need to export Openstack variables:

```
export OS_AUTH_URL=https://keystone-public-openstack.apps.ocp-4-16.standalone
export OS_PROJECT_NAME=admin
export OS_PASSWORD=admin
export OS_USERNAME=admin
export OS_DOMAIN_NAME=Default
export OS_PROJECT_ID=xyz
```

Then create the argument json file, for example:

```
cat <<EOF > args.json
{
        "user": "root",
        "password": "root",
        "server": "10.0.0.7",
        "vmname": "rhel-9.4-3",
        "cbtsync": false,
        "dst_cloud": {
            "auth": {
                "auth_url": "https://keystone-public-openstack.apps.ocp-4-
16.standalone",
                "username": "admin",
                "project_id": "xyz",
                "project_name": "admin",
                "user_domain_name": "Default",
                "password": "admin"
            },
            "region_name": "regionOne",
            "interface": "public",
            "identity_api_version": 3
        }
}
EOF
```

Then execute the `migrate` binary:

```
pushd vmware-migration-kit/vmware_migration_kit
./plugins/modules/migrate/migrate
```

You can see the logs into:

```
tail -f /tmp/osm-nbdkit.log
```

4.15. Configuration and Usage

4.16. Migration Parameters Guide

4.16.1. What are migration parameters

Resource YAML files generated by OS Migrate when exporting are editable by the user. The parameters in `params` section of a resource generally specify direct resource properties, i.e. **what** the resource is like. Sometimes, a resource can be copied from source to destination in multiple different ways, each suitable for different use cases. To control **how** an individual resource should be migrated, there is another editable section in resource serializations, called `_migration_params`. The descriptions of the most important ones are in this guide.

4.16.2. Workload migration parameters

- `boot_disk_copy` controls how if the boot disk of the destination server is copied or re-imaged:
  - `false`: The destination server will be booted from a Glance image of the same name as the source server. (This is the default for servers which were booted from an image in the source cloud.)
  - `true`: The source server's boot disk will be copied into the destination as a volume, and the destination server will be created as boot-from-volume. (For servers which are already boot-from-volume in the source cloud, this is the default and the only possible path.)
- `data_copy` controls storage migration modes for workloads:
  - `false`: The copying of data using os-migrate is skipped.
  - `true`: (Default) We result to using os-migrate for data copying.
- `boot_volume_params` controls new boot disk creation parameters in the destination, in case the source VM has no boot disk, but `boot_disk_copy` is `true`.

  There are a few pre-filled parameters, defaulted to `None`. Then `None` value means those parameters will not be specified when creating the boot volume, or in case of the `name` parameter, the default for workload migration will be used (prefix + VM name).

  When the source VM **does** have a boot volume already, do not use `boot_volume_params` to edit the destination creation parameters. Instead, edit the serialized volume in the `volumes` section of the workload's `params`.

- `boot_volume` controls new boot disk creation parameters for workload migrations with storage mode (data_copy) set to false.

  There are a few pre-filled parameters, defaulted to `None`. Then `None` value means those parameters will not be specified when creating the boot volume, or in case of the `name` parameter, the default for workload migration will be used (prefix + VM name).

  When the source VM **does** have a boot volume already, do not use `boot_volume_params` to edit the destination creation parameters. Instead, edit the serialized volume in the `volumes` section of the workload's `params`.

- `additional_volumes` any additonal volumes to be configured for workload migrations with storage mode (data_copy) set to false.

- `floating_ip_mode` controls whether and how floating IPs should be created for workloads:
    - `skip`: Do not create any floating IPs on the destination server.
    - `new`: Create a new floating IP (auto-assigned address).
    - `existing`: Assume the floating IP address as specified in the workload serialization is already assigned to the destination project, but not attached. Attach this floating IP. If this is not possible for some reason, fail.
    - `auto` (default): Attempt the `existing` method of floating IP assignment first, but should it fail, fall back to the `new` method instead.

      General remarks regarding floating IPs:

      In the `workloads.yml` export, each serialized floating IP contains a `fixed_ip_address` property, so a floating IP will be created on the port with this address. (When editing ports/fixed addresses of a workload, make sure to also edit the `fixed_ip_address` properties of its floating IPs accordingly.)

      It is important to note that the address of a newly created floating IP will be automatically selected by the cloud, it will not match the floating IP address of the source server. (In most cases, the floating IP ranges of src/dst clouds don't overlap anyway.)

4.17. Variables Guide

The goal of this document is to guide users to correctly configure the most important variables in OS Migrate. For a complete listing of variables configurable for each Ansible role, refer to the documentation of the individual roles.

4.17.1. General variables

Resource filters

Resource filters allow the user to control which resources will be migrated. The filters match against resource **names**.

The filters work **both during export and during import**, and it is not required that the same value is used during export and import. This feature can be used e.g. to export a subset of the existing resources, and then during import further limit the subset of resources being imported into batches.

The value of a filter variable is a list, where each item can be a string (exact match) or a dictionary with `regex` key (regular expression match). A resource is exported if it matches at least one of the list items.

```
os_migrate_networks_filter:
  - my_net
  - other_net
  - regex: ^myprefix_.*
```

The above example says: Export only networks named `my_net` **or** `other_net` **or** starting with `myprefix_`.

The filters default to:

```
- regex: .*
```

meaning "export all resources". (The set of resources exported will still be limited to those you can see with the authentication variables you used.)

Sometimes two roles use the same variable where this makes sense, especially for attached resources. E.g. roles `export_security_groups` and `export_security_group_rules` both use `os_migrate_security_groups_filter`. Similarly, `export_routers` and `export_router_interfaces` both use `os_migrate_routers_filter`.

List of the currently implemented filters with default values you can copy into your variables file and customize:

```
os_migrate_flavors_filter:
  - regex: .*
os_migrate_images_filter:
  - regex: .*
os_migrate_keypairs_filter:
  - regex: .*
os_migrate_networks_filter:
  - regex: .*
os_migrate_projects_filter:
  - regex: .*
os_migrate_routers_filter:
  - regex: .*
os_migrate_security_groups_filter:
  - regex: .*
os_migrate_subnets_filter:
  - regex: .*
os_migrate_users_filter:
  - regex: .*
os_migrate_workloads_filter:
  - regex: .*
```

4.17.2. Conversion host variables

The following variables are those that need to be configured prior to running OS Migrate.

Conversion host name

The conversion hosts might be configured using different names, this is in case an operator needs to have them registered with the subscription manager and avoid collisions with the names.

The conversion hosts names can be customized using the following variables:

```
os_migrate_src_conversion_host_name
os_migrate_dst_conversion_host_name
```

By default, these variables have the same value for both conversion hosts `os_migrate_conv_src` and `os_migrate_conv_dst` respectively.

Conversion host image name

The conversion host image is the guest configure to execute the instances migrations.

The variables to be configured are:

```
os_migrate_src_conversion_image_name
os_migrate_dst_conversion_image_name
```

This image must be accessible to both tenants/projects prior to executing the conversion host deployment playbook. The variables default to `os_migrate_conv`, so if a conversion host image is uploaded to Glance as public image with this name (in both src and dst clouds), these variables do not need to be configured explicitly.

Make sure this image exists in Glance on both clouds. Currently it should be a CentOS 9 Cloud Image or RHEL 8 KVM Guest Image.

Conversion host flavor name

The conversion host flavor defines the compute, memory, and storage capacity that will be allocated for the conversion hosts. It needs to have at least a volume with 20GB.

The variables to be configured are:

```
os_migrate_src_conversion_flavor_name
os_migrate_dst_conversion_flavor_name
```

Usually, 'm1.medium' will suffice this requirement, but again, it can be different between deployments.

Conversion host external network name

The external network configuration allows the connection of the conversion host router for external access, this external network must be able to allocate floating IPs reachable between both conversion hosts.

Set the name of the external (public) network to which conversion host private subnet will be attached via its router, for source and destination clouds respectively, via these variables:

```
os_migrate_src_conversion_external_network_name
os_migrate_dst_conversion_external_network_name
```

This is not required if you are attaching your conversion host to pre-existing network (when `os_migrate_src/dst_conversion_manage_network` is `false`).

Other conversion host dependency names

In addition to the name variables described above, it is possible to customize names of other conversion host dependency resources:

```
os_migrate_src_conversion_net_name
os_migrate_dst_conversion_net_name
os_migrate_src_conversion_subnet_name
os_migrate_dst_conversion_subnet_name
os_migrate_src_conversion_router_name
os_migrate_dst_conversion_router_name
os_migrate_src_conversion_secgroup_name
os_migrate_dst_conversion_secgroup_name
os_migrate_src_conversion_keypair_name
os_migrate_dst_conversion_keypair_name
```

Conversion host availablility zone management

Availability zones are defined by attaching specific metadata information to an aggregate:

```
os_migrate_src_conversion_availability_zone
os_migrate_dst_conversion_availability_zone
```

The conversion host can set logical abstractions for partitioning instances to a specific set of hosts belonging to an aggregate.

The default is `false` (meaning no specification provided).

Conversion host network management

It is possible to disable creation and deletion of conversion host private network by setting these variables to `false`:

```
os_migrate_src_conversion_manage_network
os_migrate_dst_conversion_manage_network
```

This disables creation of the network, the subnet, and the router that typically makes the conversion host reachable from outside the cloud.

When disabling network management like this, you'll need pre-existing network that the conversion host can attach to and use it to talk to the other conversion host. Set these network name variables accordingly:

```
os_migrate_src_conversion_net_name
os_migrate_dst_conversion_net_name
```

Conversion host floating IP management

OS Migrate can be told to not attempt to create any floating IPs on the conversion hosts. This can be useful when attaching a conversion host to some public network, where its IP address will be automatically reachable from outside. The variables to control whether conversion hosts should have floating IPs are:

```
os_migrate_src_conversion_manage_fip
os_migrate_dst_conversion_manage_fip
```

When the conversion hosts are removed, the required and assigned floating IPs need to be detached or removed.

The following variables allow to change the behavior of deleting of detaching the floating IP when deleting the conversion hosts (default: true):

```
os_migrate_src_conversion_host_delete_fip
os_migrate_dst_conversion_host_delete_fip
```

When the corresponding `…_manage_fip` variable is set to `false`, floating IP deletion is not attempted even if `…_delete_fip` is set to `true`.

Conversion host specific floating IP

Each conversion host needs to have a floating IP, these floating IPs can be assigned automatically or defined by the operator with the usage of the following variables:

```
os_migrate_src_conversion_floating_ip_address
os_migrate_dst_conversion_floating_ip_address
```

When using this variable to specify an exact IP address, the floating IP must already exist and be available for attaching.

Attaching conversion hosts onto public networks

A combination of variables described earlier can be used to attach the conversion hosts directly onto pre-existing public networks. We need to make sure that we don’t try to create any private network, we don’t try to create a floating IP, and we set the conversion host network names accordingly:

```
os_migrate_src_conversion_manage_network: false
os_migrate_dst_conversion_manage_network: false
os_migrate_src_conversion_manage_fip: false
os_migrate_dst_conversion_manage_fip: false
os_migrate_src_conversion_net_name: some_public_net_src
os_migrate_dst_conversion_net_name: some_public_net_dst
```

Storage migration modes

The modes for workload migrations can be changed in either cloud. The variable that control the behavior are:

```
os_migrate_workloads_data_copy
```

The default is `true` (meaning the copying of data using os-migrate is skipped).

This is useful if there are pre-created volumes in the destination cloud that we just want to attach when creating the VM in the destination.

Conversion host boot from volume

The conversion hosts can be created as boot-from-volume servers in either cloud. The variables that control the behavior are:

```
os_migrate_src_conversion_host_boot_from_volume
os_migrate_dst_conversion_host_boot_from_volume
```

The default is `false` (meaning boot from Nova local disk).

When creating boot-from-volume conversion hosts, it is possible to customize the size in GB for the boot volume:

```
os_migrate_src_conversion_host_volume_size
os_migrate_dst_conversion_host_volume_size
```

The size should be 20 or more, the default is 20.

Conversion host RHEL variables

When using RHEL as conversion host, set the SSH user name as follows:

```
os_migrate_conversion_host_ssh_user: cloud-user
```

It is also necessary to set RHEL registration variables. The variables part of this role are set to `omit` by default.

The variables `os_migrate_conversion_rhsm_auto_attach` and `os_migrate_conversion_rhsm_activationkey` are mutually exclusive, given that, they are both defaulted to omit.

Typically the only registration variables to set are:

```
os_migrate_conversion_rhsm_username
os_migrate_conversion_rhsm_password
```

In this case, `os_migrate_conversion_rhsm_auto_attach` should be set to `True` in order to fetch automatically the content once the node is registered.

or:

```
os_migrate_conversion_rhsm_activationkey
os_migrate_conversion_rhsm_org_id
```

For this case, `os_migrate_conversion_rhsm_auto_attach` must be left undefined with its default value of `omit`.

The complete list of registration variables corresponds to the redhat_subscription Ansible module. In OS Migrate they are named as follows:

```
os_migrate_conversion_rhsm_activationkey
os_migrate_conversion_rhsm_auto_attach
os_migrate_conversion_rhsm_consumer_id
os_migrate_conversion_rhsm_consumer_name
os_migrate_conversion_rhsm_consumer_type
os_migrate_conversion_rhsm_environment
os_migrate_conversion_rhsm_force_register
os_migrate_conversion_rhsm_org_id
os_migrate_conversion_rhsm_password
os_migrate_conversion_rhsm_pool
os_migrate_conversion_rhsm_pool_ids
os_migrate_conversion_rhsm_release
os_migrate_conversion_rhsm_rhsm_baseurl
os_migrate_conversion_rhsm_rhsm_repo_ca_cert
os_migrate_conversion_rhsm_server_hostname
os_migrate_conversion_rhsm_server_insecure
os_migrate_conversion_rhsm_server_proxy_hostname
os_migrate_conversion_rhsm_server_proxy_password
os_migrate_conversion_rhsm_server_proxy_port
os_migrate_conversion_rhsm_server_proxy_user
os_migrate_conversion_rhsm_syspurpose
os_migrate_conversion_rhsm_username
```

Additionally is possible to enable specific repositories in the conversion hosts using the following variable:

```
os_migrate_conversion_rhsm_repositories
```

The `os_migrate_conversion_rhsm_repositories` variable is a list of those repositories that will be enabled on the conversion host.

Enabling password-based SSH access to the conversion hosts

When required, a user can configure password-based SSH access to the conversion hosts, this feature might be useful for debugging when the private key of the hosts is not available anymore.

The variables required in order to configure the password-based access are named as follows:

```
os_migrate_conversion_host_ssh_user_enable_password_access
os_migrate_conversion_host_ssh_user_password
```

The variable `os_migrate_conversion_host_ssh_user_enable_password_access` is set by default to `false`, and the variable `os_migrate_conversion_host_ssh_user_password` is set by default to the following string `weak_password_disabled_by_default`.

The user enabled to access the conversion hosts with password-based authentication is the one defined in the `os_migrate_conversion_host_ssh_user` variable.

Running custom bash scripts in the conversion hosts

It is possible to run custom bash scripts in the conversion hosts before and after configuring their content. The content of the conversion hosts is a set of required packages and in the case of using RHEL then the configuration of the subscription manager.

The variables allowing to run the custom scripts are:

```
os_migrate_src_conversion_host_pre_content_hook
os_migrate_src_conversion_host_post_content_hook
os_migrate_dst_conversion_host_pre_content_hook
os_migrate_dst_conversion_host_post_content_hook
```

The Ansible module used to achieve this is shell, so users can execute a simple one-liner command, or more complex scripts like the following examples:

```
os_migrate_src_conversion_host_pre_content_hook: |
  ls -ltah
  echo "hello world"
  df -h
```

or:

```
os_migrate_src_conversion_host_pre_content_hook: "echo 'this is a simple command'"
```

Disabling the subscription manager tasks

It is possible to disable the subscription manager native tasks by setting to false the following variable:

```
os_migrate_conversion_rhsm_manage
```

This will skip the tasks related to RHSM when using RHEL in the conversion hosts. Disabling RHSM can be useful in those cases where the operator has custom scripts they need to use instead the standard Ansible module.

4.17.3. OpenStack REST API TLS variables

If either of your clouds uses TLS endpoints that are not trusted by the Migrator host by default (e.g. using self-signed certificates), or if the Migrator host should authenticate itself via key+cert, you will need to set TLS-related variables.

- `os_migrate_src_validate_certs` / `os_migrate_dst_validate_certs` - Setting these to `false` disables certificate validity checks of the source/destination API endpoints.
- `os_migrate_src_ca_cert` / `os_migrate_dst_ca_cert` - These variables allow you to specify a custom CA certificate that should be used to validate the source/destination API certificates.
- `os_migrate_src_client_cert`, `os_migrate_src_client_key` / `os_migrate_dst_client_cert`, `os_migrate_dst_client_key` - If the Migrator host should authenticate itself using a TLS key certificate when talking to source/destination APIs, set these variables.

4.17.4. Workload import/export variables

- `os_migrate_workload_stop_before_migration` - Set to true if you wish for os_migrate to stop your workloads/vms prior to migration. Note that only workloads/vms in `SHUTOFF` state will be migrated.

4.17.5. Workload migration variables

Workloads to be migrated with OS Migrate can have varying storage configurations in the source cloud, and the desired way to migrate their storage also varies, per cloud operators preference.

The following table summarizes the matrix of options (whats in the source, how it should be migrated, how should OS Migrate workloads YAML file be configured, is the conversion host required for this mode of migration, is this migration mode implemented).

[Screenshot 2024-07-15 at 2 59 17 PM] | *https://github.com/user-attachments/assets/1862b21b-4f67-47c0-ba73-f62df0d4568a*

Figure 1. Screenshot 2024-07-15 at 2 59 17 PM

[Screenshot 2024-07-15 at 3 02 28 PM] | *https://github.com/user-attachments/assets/939c98fb-f425-4f53-aca4-fd03f111fd33*

Figure 2. Screenshot 2024-07-15 at 3 02 28 PM

[Screenshot 2024-07-15 at 3 03 16 PM] | *https://github.com/user-attachments/assets/faf09224-fb11-417e-865a-72c9936bc8bf*

Figure 3. Screenshot 2024-07-15 at 3 03 16 PM

4.18. General usage notes

- Run against **testing/staging clouds first**, verify that you are getting the expected results.
- Use **different `os_migrate_data_dir` per project** you're authenticating to. OS Migrate works in project (tenant) scope most of the time. The data dir will be populated with the source project's exported resources, and should not be mixed with another project's resources.

  When you are changing `os_migrate_src_auth` or `os_migrate_src_region_name` parameters, make sure to also change `os_migrate_data_dir`.
- Use the **same version of OS Migrate for export and import**. We currently do not guarantee that data files are compatible across versions.
- OS Migrate may not fit each use case out of the box. You can craft custom playbooks using the OS Migrate collection pieces (roles and modules) as building blocks.
- OS Migrate has supported migrations for OSP versions 13 to 16, 16 to 18, and simple upgrades from 15 to 16 or 16 to 17. We have tested migrations between 13 to 16, 16 to 18 using RHEL 8.

4.19. Maintenance and Support

4.20. Troubleshooting

4.20.1. General tips

- Run `ansible-playbook` with `-v` parameter to get more detailed output.

4.20.2. Common issues

- `DataVersionMismatch: OS Migrate runtime is version 'X.Y.Z', but tried to parse data file 'abc.yml' with os_migrate_version field set to 'A.B.C'. (Exported data is not guaranteed to be compatible across versions. After upgrading OS Migrate, make sure to remove the old YAML exports from the data directory.)`

  When OS Migrate export playbooks run, the existing data files aren't automatically truncated. OS Migrate gradually adds each resource serialization to the (perhaps existing) YAML file, or it updates a resource serialization if one with the same ID is already present in the file.

  OS Migrate will refuse to parse YAML files which were created with a different version. In many cases such parsing would "just work", but not always, so OS Migrate is being defensive and requires clearing out the data directory when upgrading to a new version, and re-running the export playbooks.

Alternatively, an advanced user can verify that the previous and new OS Migrate does not include any change in export data structures, and can edit the `os_migrate_version` field in the data files. This option should be used with caution, but it may prove useful in special scenarios, e.g. if external causes prevent re-exporting the data.

- `AnsibleCensoringStringIssue: workloads.yml setup task altering log_file path during preliminary import workload steps. (As a result susequent import tasks are failing due to non-existent path error.)`

  OS Migrate uses OpenStack modules to build their argument spec by using a function in OpenStack module utils. When project names are marked as `no_log` it causes values to be censored in the response. This is seen here in the import workloads setup task where `/home/project_name/workloads/import_workloads.yml` becomes `/home//workloads/import_workloads.yml`.

  OS Migrate cannot specify that only the password in the credentials dictionary should be treated as a secret, instead the whole credentials dictionary is marked as a secret.

  A workaround to this is to sanitize the project name with something in a pre-migration playbook that sets up storage directories for OS Migrate variables or data. This can prove beneficial in the event of users running into censored string issues relating to ansible.

- `KeypairMigrationPitfalls: Keys are not seen by user performing migrations. When a user creates keypairs and assigns those keypairs to its inteded resource its noted that users used in the migration process can access the inteded resources but not the required keys. This leads to checks failing since the user can’t check if the key exist in the source cloud.`

  OS Migrate has an export_user_keypairs.yml which escalates using admin privileges. By default it iterates over all users and their keys, but it listens for filter variables which can help scope down the export.

  How to use those key exports depends on how the workload migration should be done. Either the keys can be uploaded to destination to the respective users via import_users_keypairs.yml playbook, and destination credentials for workload migration have to be of the users who can see the keys.

  An alternative for the following issues is the user_ref.name and user_ref.domain_name in the exported YAML could be edited from actual names to %auth% values, and that data file could then be used with import_keypairs.yml (run as a tenant user, not admin), which would import all the various keys under a single user, and that user could then perform the migration, having all the necessary public keys.

4.20.3. Upgrading

This document describes the recommended method of upgrading OS Migrate from Ansible Galaxy.

Collection upgrade

To upgrade to the latest release if you already have the OS Migrate collection installed, make sure to pass the `-f` flag to the installation command, which will force installing the (latest) collection even if it is already present:

```
ansible-galaxy collection install -f os_migrate.os_migrate
```

To upgrade/downgrade to a particular release:

```
ansible-galaxy collection install os_migrate.os_migrate:<VERSION>
```

You can find available releases at OS Migrate Galaxy page.

Usage notes related to upgrading

- OS Migrate presently does not guarantee any forward compatibility of exported data. **The same version of OS Migrate should be used during export and import.**
- After upgrading, **clear any potential existing data files** from your `os_migrate_data_dir`, or use a different one.

  During export, OS Migrate will attempt to parse exsiting data files (with the intention of adding new resources to them), and an error will be raised if the existing data files were created with a different OS Migrate version.

4.21. OS Migrate Contribution Guidelines

4.21.1. TBD - but largely based on combination of existing guide, and David & Roberto collaboration

4.22. Reference

4.23. OS Migrate Glossary

A comprehensive glossary of terms for operators learning about the OS Migrate project.

4.23.1. Core Concepts

OS Migrate Collection

An Ansible collection (`os_migrate.os_migrate`) that provides modules, roles, and playbooks for migrating OpenStack resources between clouds.

Resource

An OpenStack entity that can be migrated (networks, instances, flavors, images, etc.). Each resource type has its own export/import workflow.

Export Phase

The process of extracting resource definitions from a source OpenStack cloud and serializing them to YAML files.

Import Phase

The process of reading YAML resource files and creating corresponding resources in a destination OpenStack cloud.

Parallel Migration

A modernization strategy where a new OpenStack deployment runs alongside an existing one, with tenant resources migrated between them.

Idempotent Operations

All playbooks can be re-run safely; they won’t duplicate resources or cause conflicts.

4.23.2. Resource Types

Workloads

Running instances/VMs that are migrated from source to destination cloud, including their attached storage and network configuration.

Detached Volumes

Storage volumes not currently attached to any instance.

Flavors

Virtual machine templates that define CPU, memory, and disk specifications.

Images

Disk images used as templates for creating instances.

Networks

Virtual networks that provide connectivity between instances.

Subnets

IP address ranges within networks that define available IP addresses.

Routers

Virtual routers that provide connectivity between networks and external networks.

Router Interfaces

Connections between routers and subnets.

Security Groups

Firewall rule sets that control network traffic to instances.

Security Group Rules

Individual firewall rules within security groups.

Projects

OpenStack tenants that contain and isolate resources.

Users

OpenStack user accounts with authentication credentials.

Keypairs

SSH key pairs used for secure instance access.

User Project Role Assignments

Mappings that grant users specific roles within projects.

4.23.3. Migration Infrastructure

Conversion Host

A temporary VM created in the destination cloud to facilitate workload migration, particularly for disk image transfer.

Conversion Host Content

Software and configuration deployed on conversion hosts to enable migration functionality.

Migrator Host

The system where OS Migrate playbooks are executed, coordinating the migration process.

4.23.4. Data Management

Data Directory (`os_migrate_data_dir`)

Local filesystem location where exported YAML resource files are stored during migration.

Resource Filter (`os_migrate_<resource>_filter`)

Name-based filtering to selectively export/import specific resources rather than all resources of a type.

Serialization

The process of converting OpenStack SDK objects into OS Migrate’s standardized YAML format.

Resource Validation

Checking that imported YAML files contain valid resource definitions before attempting import.

File Validation (`import_<resource>_validate_file`)

Optional validation step that verifies resource data integrity before import operations.

4.23.5. Resource Structure

params_from_sdk

Resource properties that are copied to the destination cloud (configuration, settings).

info_from_sdk

Resource properties that are NOT copied (UUIDs, timestamps, read-only data).

Migration Parameters

OS Migrate-specific settings that control migration behavior for each resource.

SDK Parameters

Parameters used when making OpenStack API calls to create or update resources.

Readonly SDK Parameters

Parameters that are allowed during resource creation but not during updates.

4.23.6. Authentication & Configuration

Source Auth (`os_migrate_src_auth`)

OpenStack authentication credentials for the source cloud.

Destination Auth (`os_migrate_dst_auth`)

OpenStack authentication credentials for the destination cloud.

clouds.yaml

OpenStack client configuration file containing cloud authentication details.

Source Cloud (`SRC_CLOUD`)

Environment variable identifying the source cloud configuration in clouds.yaml.

Destination Cloud (`DST_CLOUD`)

Environment variable identifying the destination cloud configuration in clouds.yaml.

4.23.7. Ansible Components

Export Roles

Ansible roles that call export modules and handle resource filtering (e.g., `export_networks`, `export_workloads`).

Import Roles

Ansible roles that validate data files and call import modules (e.g., `import_networks`, `import_workloads`).

Export Modules

Ansible modules that retrieve resources from source cloud and serialize to YAML (e.g., `export_flavor.py`).

Import Modules

Ansible modules that read YAML files and create resources in destination cloud (e.g., `import_flavor.py`).

Prelude Roles

Setup roles that prepare the environment:

- `prelude_src` - Source cloud preparation
- `prelude_dst` - Destination cloud preparation
- `prelude_common` - Common setup tasks

Resource Class

Python class that defines how OpenStack SDK objects are converted to/from OS Migrate format. All inherit from the base `Resource` class.

4.23.8. Development & Testing

Role Skeleton

Template structure for creating new export/import roles using the role-addition script.

Unit Tests

Tests that verify module logic and resource class behavior in isolation.

Functional Tests

Tests that verify roles and modules work correctly in integration scenarios.

End-to-End Tests (E2E)

Full migration workflow tests that validate complete export/import cycles.

Sanity Tests

Ansible collection sanity checks that verify code quality and standards compliance.

4.23.9. Container Environment

Container Engine

Configurable container runtime (Podman/Docker) used for development environment via `CONTAINER_ENGINE` variable.

Development Container

CentOS Stream 10 container with Python 3.12 where all OS Migrate commands execute.

Source Mount

The `/code` directory inside the container where the OS Migrate source code is mounted.

4.23.10. Build & Deployment

Collection Build

Process of packaging OS Migrate into an Ansible collection archive for distribution.

Galaxy Installation

Installing the OS Migrate collection via `ansible-galaxy collection install` `os_migrate.os_migrate`.

Dependencies

Required Ansible collections (`community.crypto`, `community.general`, `openstack.cloud`) and Python packages.

4.23.11. Migration Process

Three-Phase Migration

The standard workflow: Export → Transfer → Import.

Transfer Phase

Moving exported YAML data files from source environment to destination environment.

Validation Phase

Verifying that resources were created correctly and handling any migration errors.

Resource References

Cross-references between resources (e.g., instances referencing networks) that must be resolved during import.

---

[1] CBT = https://knowledge.broadcom.com/external/article/320557/changed-block-tracking-cbt-on-virtual-ma.html